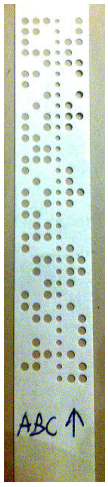
Proceedings of the Society of PoC || GTFO
Issue 0x01, an Epistle to the 10th H2HC in São Paulo

From the writing desk, not the raven, of Rt. Revd. Preacherman Pastor Manul Laphroaig
pastor@phrack.org

October 6, 2013

1 Call to Worship

Neighbors, please join me in reading this second 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 Issue 0x00, we politely suggest pirating it from the usual locations, or on paper from a neighbor who picked up a copy in Vegas.



In Section 2, Dan Kaminsky presents of all strange things a *defensive* PoC! His four lines of Javascript seem to produce random bytes, but that can't possibly be right. If you disagree with him, POC||STFU.

This issue's devotional is in Section 3, where Travis Goodspeed shares a thought experiment in which Ada Lovelace and Serena Butler fight on opposite sides of the Second War on General Purpose Computing using Don Lancaster's TV Typewriter as ammunition.

In the grand tradition of backfiring parse tree differentials, Ange Albertini shares in Section 4 a nifty trick for creating a PE file that is interpreted differently by Windows XP, 7, and 8. Perhaps you'll use this as an anti-reversing trick, or perhaps you'll finally learn why TinyPE doesn't work after XP. Either way, neighborliness abounds.

In Section 5, Julia Wolf demonstrates on four napkins how to make a PDF that is also a ZIP. Perhaps, dear reader, if you are reading this from a PDF you might find a file or two to be attached?

In Section 6, Josh Thomas will teach you a how to permanently brick an Android phone by screwing around with its voltage regulators in quick kernel patch. We the editors remind readers to send only quality, technical correspondence to Josh; any rubbish that merely advocates your chosen brand of cellphone should be sent to jobs@paper.li.

Today's sermon, to be found in Section 7, concerns the divinity of programming languages, from PHP to BASIC. Following along with a little scripture and a lot of liquor, we'll see that every language has a little something special to make it worth learning and teaching. Except Java.

Finally, in Section 8, we pass the collection plate and beg that you contribute some PoC of your own. Articles should be short and sweet, written such that a reader will be inspired to build something clever.

This issue is dedicated to the continuing ministry of Mitch Altman, a Johnny Appleseed of soldering literacy who has taught countless adults and countless children in countless cities to build their own electronics.

2 Four Lines of Javascript that Can't Possibly Work So why do they?

by Dan Kaminsky

```
// These functions form an RNG.
function millis()          {return Date.now();}
function flip_coin()       {n=0; then = millis()+1; while(millis()<=then) {n=!n;} return n;}
function get_fair_bit()    {while(1) {a=flip_coin(); if(a!=flip_coin()) {return(a);}}}
function get_random_byte(){n=0; bits=8; while(bits--){n<<=1; n|=get_fair_bit();} return n;}

// Use it like this.
report_console = function() {while(1) {console.log(get_random_byte());}}
report_console();
```

2.1 Introduction

When Apple's iPhone 5S was announced, a litany of criticism against its fingerprint reader was unleashed. Clearly, it would be vulnerable to decade old gelatin cloning attacks. Or clearly, it would utilize subdermal analysis or electrical measurement or liveness checking and not be vulnerable at all. Both fates were possible.

It took Nick DePetrillo and Rob Graham to say, "PoC || GTFO."

What Starbug eventually demonstrated was that the old attacks do indeed still work. It didn't have to be that way, but at the heart of science is experimentation and testing. The very definition of unscientific work is not merely that it will not be subjected to test but that by design it cannot.

Of course, I am not submitting an article about the iPhone 5S. I'm here to write about a challenge that's been quietly going on for the last two years, one that remains unbroken.

Can we use the clock differentials, baked into pretty much every piece of computing equipment, as a source for a True Random Number Generator? We should find out.

2.2 Context

"The generation of random numbers is too important to be left to chance," as Robert R. Coveyou from Oak Ridge liked to say. Computers, at least as people like to mentally model them, are deterministic devices. The same input will always lead to the same output.

Electrically, this is unnecessary. It takes a lot of work to make an integrated circuit completely reliable. Semiconductors are more than happy to behave unpredictably. Semiconductor manufacturers, by contrast, have behaved very predictably, refusing to implement what would admittedly be a rather difficult part to test.

Only recently have we gotten an instruction out of Intel to retrieve random numbers, RDRAND. I can't comment as to the validity of the function except to say that any audit process that refuses its auditors physical access to the part in question and disables all possible debugging or post-verification after release is not a process that inspires confidence.

But do we need the instruction? The core assumption is that in lieu of RDRAND the computer is deterministic, that the same input will lead to the same output. Seems reasonable, until you ask:

If all I do is turn a computer on, will it take the same number of nanoseconds to reach the boot screen?

If you think the answer is yes, PoC || GTFO.

If you think the answer is no, that there will be some amount of nanosecond drift, then where does this drift come from? The answer is that the biggest lie about your computer is that it's just one computer. CPU cores talk to memory busses talk to expansion busses talk to storage and networking and the interrupt of the month club. There are generally some number of clocks, they have different speeds and different tolerances, and you do not get them synchronized for free. (System-on-Chip devices are a glaring exception, but it's still rather common for them to be speaking to peripherals.)

Merely turning the machine on does not synchronize everything, so there is drift. Where there is drift, there is entropy. Where there is entropy, there is security.

2.3 This is Actually a Problem

To stop a brute force attack against your random number generator, you need a few bits. At least 80, ideally 128. Not 128 million. 128. Ever. For the life of that particular device. (Not model! The attacker can just go out and buy one of those devices, and find those 128 bits.) Now you may say, "We need more than 128 bits for production." And that's fine. For that, we have what are known as Cryptographically Secure Pseudo Random Number Generators (CSPRNG's). Seed 128 bits in, get an infinite keystream out. As long as the same seed is never repeated, all is well.

Cryptographers love arguing about good CSPRNGs, but the reality is that it's not that hard to construct one. Run a good cipher or hash function (not RC4) in pretty much any sort of loop and the best attack reduces to breaking that cipher or hash function. (If you disagree, PoC ‖ GTFO.) That's not to say there aren't "nice to have" properties that an ideal CSPRNG can acquire, but empirically two things have actually happened in the real world some of us are trying to defend.

First, most PRNG's aren't cryptographically secure. Most random numbers are not securely generated. They could be. CSPRNGs can certainly be fast enough. If we really wanted, they could be simple enough too. To be fair, the advice of "Just use /dev/urandom." is what most languages should follow. But there's a second issue, and it's severe.

The second issue, the hard part, is not expanding 128 bits to an infinite stream. The hard part is actually getting those 128 bits! So called "True Random Number Generation" is actually the thing we are bad at, in the real world. The CSPRNG of the gods falls to a broken TRNG. What is a kernel supposed to do when /dev/urandom wants data and there is no seed? The whole idea behind /dev/urandom is that it will provide answers immediately. And so, in general, it does.

And then Nadia Heninger scans the Internet, and finds that 1/200 RSA keys are badly formed. That's a floor, by the way. Keys that are similar but not quite identical are not counted in that 1/200. But of course, buying a handful of devices gives you the similarity map.

However bad clock differentials might be, they would not have created this apocalyptic failure rate.

2.4 This Didn't Have to Happen

In 1999, Daniel J. Bernstein pointed out that the 16 bit transaction ID in DNS was insufficient and that the UDP source port could be overloaded to provide almost 32 bits of entropy per DNS request. His advice was not accepted.

In 1996, Matt Blaze created Truerand, a scheme that pitted the CPU against signal handlers. His approach actually has a long and storied history, back to the VMS days, but it was never accepted either.

In 2011, I released Dakarand. Dakarand is a collection of approaches for pitting various clocks inside against a computer against each other. Many random number generation schemes come down to measuring something that varies by millisecond with something that varies by nanosecond. (Your CPU, running in a

tight loop, is a fast clock operating in the gigahertz. Your RTC—Real Time Clock—is much slower and is not reporting milliseconds accurate to the nanosecond. In confusion, profit.)

Dakarand may in fact fail, somehow, somewhere, in some mode. But thus far, it seems to work pretty much everywhere, even virtual machines. (As a TRNG, each read event can generate new seed material without depending on data that might have been inherited before VM cloning.)

In 2013, in honor of Barnaby Jack, I tossed together the code at the top of this article. It's the weakest possible formulation of this concept, written in JavaScript and hardened only with the barest level of Von Neumann. It is called oi.js, and you should break it.

After all, it's just JavaScript. It can't be secure.

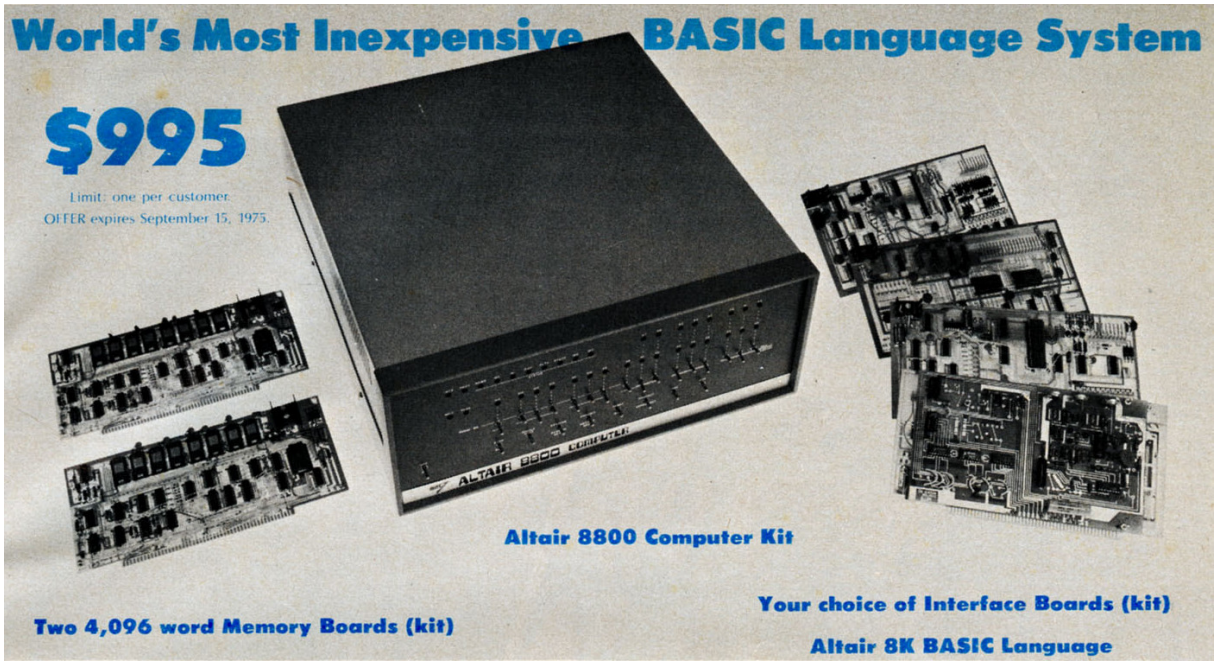
The idea is, in fact, to find the weakest formulation of this concept that still works. PoC ‖ GTFO shows us where known security stops and safety margin begins.

2.5 On Measuring the Strength of Cryptosystems

Sometimes people forget that we regularly build remarkably safe code out of seemingly trivial to break components. Hash functions are generally composed of simple operations that, with only a few rounds of those functions, start becoming seriously tricky to reverse. RSA, through this lens, is just multiply as an encryption function, albeit with a mind bending number of rounds.

Humans do not require complex radioactivity measurements or dwellings on the nature of the universe to get a random bit. They can merely flip a coin, a system that is well described as the Newtonian interaction between a slow clock (coin goes up, coin goes down) and a fast clock (coin spins round and round.) Pretending that there is nothing with the properties of a simple coin anywhere in the mess that is a device that can at least run Linux is enabling vulnerability.

PoC's in defense are rare—now let's see what you've got ;)



3 Weird Machines from Serena Butler's TV Typewriter

by Travis Goodspeed

In the good old days, one could make the argument–however fraudulent!–that memory corruption exploits were only used by the bad guys, to gain remote code execution against the poor good guys. The clever folks who wrote such exploits were looked upon as if they were kicking puppies, and though we all knew there was a good use for that technology, we had little more than RMS's paranoid ramblings about fascism to present as a legitimate use-case. Those innocent days in which exploit authors were derided as misfits and sinners are beginning to end, as children must now use kernel exploits to program their own damned cell phones. If we as authors of weird machines are to prepare for the future, it might be a good idea to work out a plan of last resort. What could be build if computers themselves were outlawed?

I'm writing to share with you the concept of a Butlerian Typewriter, loosely inspired by Cory Doctorow's 28C3 lecture and strongly inspired by many good nights of fine scotch with Sergey Bratus, Meredith Patterson, Len Sassaman, Bx Shapiro, and Julian Bangert. It's a little thought experiment about what weird machines could be constructed in a world that has outlawed Turing-completeness.

In the universe of Frank Herbert's Dune, the war on general-purpose computing is over, and the computers lost—but not before they struck first, enslaved humanity, and would have eliminated it if it were not for one Serena Butler. St. Serena showed the way by defenestrating a robotic jailer, leading the rest of humanity in the Butlerian Jihad against computers and thinking machines. Having learned the hard way that building huge centralized systems to run their lives was not a bright idea, humans banned anything that could grow into one.

So general-purpose computers still exist on the black market, and you can buy one if you have the right connections and freedom from prosecution, but they are strictly and religiously illegal to possess or manufacture. The Orange Catholic Bible commands, "Thou shalt not make a machine in the likeness of a man's mind."

Instead of general purpose computers, Herbert's society has application-specific machines for various tasks. Few would argue that a typewriter or a cat picture are dangerous, but your iPhone is a heresy. Siri would be mistaken for the Devil himself.

Let's simplify this rule to Turing-completeness. Let's imagine that it is illegal to possess or to manufacture a Universal Turing Machine. This means no ELF or DWARF interpreters, no HTML5 browsers. No present-day CPU instruction set is legal either; not ARM, not MIPS, not PowerPC, not X86, and not AMD64. Not even a PDP11 or MSP430. Pong would be legal, but Ms. Pac-Man would not. In terms of Charles Babbage's work, the Difference Engine would be fine but the Analytical Engine would be forbidden.

Now comes the fun part. Let's have a competition between Ada Lovelace and Serena Butler. Serena's goal is to produce what we will call a Bulterian Typewriter, an application-specific word processor of sorts. She can use any modern technology in designing the typewriter, as such things are available to her from the black market. She even has access modern manufacturing technology, so producing microchips is allowed if they are not Turing-complete. She may not, however, produce anything contrary to the O.C.B.'s prohibition against thinking machines. Nothing Turing-complete is legal, and even her social standing isn't sufficient to get away with mass production of computers.

So Serena designs a Butlerian Typewriter using black market tools like Verilog or VHDL, then mass produces it for release on the white market as a consumer appliance with no Turing machine included. One might imagine that she would begin with a text buffer, wiring its output to

a 1970's cathode-ray television and its input to a keyboard. Special keys could navigate through the buffer. Not very flashy by comparison to today's tweety-boxes, but it can be done.

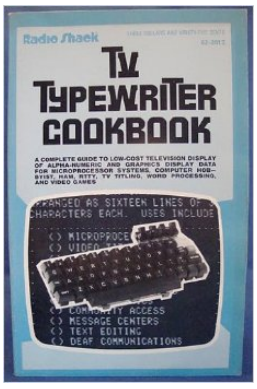
After this typewriter hits the market, Ada Lovelace comes into play. Ada's unpaid gambling debts prevent her from buying on the black market, so she has no way to purchase a computer. Instead, her goal is to build a computer from scratch out of the pieces of a Butlerian Typewriter. This won't be easy, but it's a hell of a lot simpler than building a computer out of mechanical disks or ticker-tape!

In playing this as a game of conversation with friends, we've come to a few conclusions. First, it is possible for Serena to win if (1) she's very careful to avoid feature creep, (2) the typewriter is built with parts that Ada cannot physically rewire, and (3) Ada only has a single machine to work with. Second, Ada seems to always win (1) if the complexity of the typewriter passes a certain threshold, (2) if she can acquire enough typewriters, or (3) if the parts are accessible enough to rewire.

As purpose of the game is to get an intuitive feeling for how to build computers out of twigs and mud, let's cover some of the basic scenarios. (The game is little fun when Serena wins, so her advocate almost always plays both sides.)

- If Serena builds her machine from 7400-series chips, Ada can rewire those chips into a general-purpose computer.
- If Ada can purchase thousands of typewriters, she can rewire each into some sort of 7400-equivalent, like a NAND gate. These wouldn't be very power-efficient, but Ada could arrange them to form a computer.
- If Serena adds any sort of feedback from the output of the machine to the input, Ada gets a lot more room to maneuver. Spellcheck can be added safely, but storage or text justification is dangerous.
- It's tempting to say that Serena could win by having a mask-programmed microcontroller that cannot execute RAM, but software bugs will likely give a victory to Ada in this case. This is only interesting because it's the singular case where academics' stubborn insistence that ROP is different from ret-to-libc might actually be relevant!

So how does a neighbor learn to build these less-than-computers, and how does another neighbor learn to craft computers out of them? If you are unfamiliar with hardware design languages, start off with a tutorial in VHDL or Verilog, then work your way up to crafting a simple CPU in the language. After that, sources get a bit harder to come by.



A primitive sort of Bulterian Typewriter is described by Don Lancaster in his classic article TV Typewriter from the September 1973 issue of Radio Electronics. His follow-up book, the TV Typewriter Cookbook, is as complete a guide you could hope for when designing these sorts of machines. Don Lancaster's book as well as his article are available for free on his website, but you'd do well to spend 15¢ on a paperback from Amazon.

Lancaster's TV Typewriter differs from Serena's in a number of ways, but chief among them is motivation. He avoided a CPU because he couldn't afford one, and he limited RAM because it was hellishly expensive in 1973. By contrast, Serena is interested in building what a brilliant engineer like Don might have made with today's endless quantities of memory and modern ASIC fabrication, while still avoiding the CPU and hoping to avoid Turing-completeness entirely.

In addition to Lancaster's book, those wishing to learn more about how to build fancy electronics without computers should buy a copy of How to Design & Build Your Own Custom TV Games by David L. Heiserman.

Published in 1978, the book is still the best guide to building interactive games around substantially analog components. For example, he shows how the paddles in a table-tennis game can be built from 555 timers, with the controllers being variable resistors that increase or decrease the time from the page blank to the drawing of the paddle.

To get some ideas for building computers out of twigs and mud, take a look at the brilliant papers by Dartmouth's Scooby Crew. They've built thinking machines from DWARF,[1] ELF,[2] and even the X86 MMU![3] I fully expect that by the end of the year, they'll have built a Turing-machine from Lancaster's original 1973 design.

Let's take a look at some examples of these fancy typewriters. I hope you will forgive me for asking annoying questions for each, but still more, I hope you will argue over each question with a clever neighbor who disagrees.

Simple BT: As a starting point, the simplest form of a Butlerian Typewriter might consist of a Keyboard that feeds into a Text Buffer that feeds into a Font ROM that feeds into an NTSC Generator that feeds into an analog TV. The Text Buffer would be RAM alternately addressed by the keyboard on the write phase and a line/row counter on the read phase. As the display's electron beam moves left to right, individual letters are fetched from the appropriate row of the Text Buffer and used as an address in the Font ROM to paint that letter on the screen.

This is roughly the sort described in Lancaster's original article. Note that it does not have storage, spell-check, justification, I/O, or any other fancy features, although he describes a few such extensions in his TV Typewriter Cookbook.

BT with Storage: There are a few different ways to implement storage. The simplest might be for Serena to battery-back the character buffer and have it as a removable cartridge, but that exposes the memory bus

[1] Exploiting the Hard Working Dwarf from WOOT 2011
[2] "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata from WOOT 2013
[3] Page Fault Liberation Army from 29C3

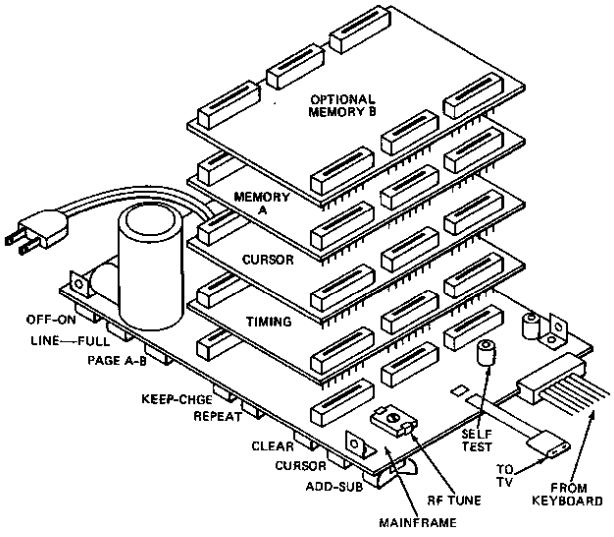


Figure 1: Don Lancaster's 1973 TV Typewriter

to Ada's manipulations. It's not hard to rewire a parallel RAM chip to be a logic gate by making its data a lookup table; this is how the first FPGA cells operated.

So if a removable memory isn't an option, what is? Perhaps Serena could make a removable typewriter module that holds everything but the keyboard, but that wouldn't allow for the copying of documents. Serial memory, such as an SPI Flash or EEPROM chip, is a possibility, but there's no good reason to think that it's any safer than parallel RAM.

A pessimist might say that external storage is impossible unless Ada is restricted to a small number of typewriters, but there's a loophole nearly as old as Mr. Edison himself. The trick is to have the typewriter flush its buffer to an audio cassette through a simple modem, and you'll find handy schematics for doing just that in Lancaster's book. Documents can be copied, or even edited, by splicing the tape in an old-fashioned recording studio.

Why is it that storage to an audio cassette is safer than storage to a battery-backed RAM module? At what point does a modem and tape become the sort of tape that Turing talked about?

BT with Spellcheck: Let's consider the specific case in which Serena has a safe design of a minimal typewriter and wishes to add spell check. The trick here is to build a hardware associative memory with a ROM that contains the dictionary. As the display's electron beam moves left to right, the current word is selected by division on spaces and newlines, and fed into the Spellcheck ROM, a hardware associative memory containing a list of valid words. The output of this memory is a single bit, which is routed to the color input of the NTSC Generator. With matching words in white and suspicious words in red, the typewriter could behave much like emacs' flyspell-mode.

So long as the associative memory is in ROM, this seems like a rather safe addition. What sort of dangers would be introduced if the associative spellcheck dictionary were in RAM? How difficult would it be to build a CPU from nothing but a few associative memory units, if you had direct access to their bus but could not change any internal wiring? How few memories would you need?

BT with Printing: Printing turns out to be much easier than electronic storage. The first method is to simply expose photographic film to the display, much as oscilloscopes were photographed in the good ol' days.

Another method would be to include a daisy wheel, dot matrix, or thermal print-head fed by a different Font ROM at a much slower scan rate. While much more practical than taking a dozen Polaroid photographs, it does give Ada a lot more room to work with, as the wiring would be exposed for her to tap and rewire.

I don't expect general purpose computing to be outlawed any time soon, but I do expect that the days of freely sharing software might soon be over. At the same time that app stores have ruthlessly killed the shareware culture that raised me as a child, it's possible that someday exploit mitigations might finally kill off remote code execution.

At the same time that we fight the good fight by developing new and clever mitigation bypasses, we ought to develop new and clever ways to build computers out of whatever scraps are left to us when straight-jacketed in future consumer hardware. Without Java, without Flash, without consistent library locations, without predictable heap allocations, our liquored and lovely gang continues to churn out exploits. Without general-purpose computing, could we do the same?

Please share this article with a neighbor,
and also share a bottle of scotch,
and argue in the kitchen for hours and hours,
—Travis

4 Making a Multi-Windows PE

by Ange Albertini

4.1 Evolution of the PE Loader

The loader for PE, Microsoft's *Portable Executable* format, evolved slowly, and became progressively stricter in its interpretation of the format. Many oddities that worked in the past were killed in subsequent loader versions; for example, the notorious TinyPE[4] doesn't work after Windows XP, as subsequent revisions of Windows require that the `OptionalHeader` is not truncated in the file, thus forcing a TinyPE to be padded to 252 bytes (or 268 bytes in 64 bit machines) to still load. Windows 8 also brings a new requirement that $AddressOfEntryPoint \leq SizeOfHeaders$ when $AddressOfEntryPoint \neq 0$, so old-school packers like FSG[5] no longer work.

So there are many real-life examples of binaries that just stop working with the next version of Windows. It is, on the other hand, much harder to create a Windows binary that would continue to run, but differently—and not just because of some explicit version check in the code, but because the loader's interpretation of the format changed over time. This would imply that Windows is not a single evolving OS, but rather a succession of related yet distinct OSes. Although I already did something similar, my previous work was only able to differentiate between XP and the subsequent generations of Windows.[6] In this article I show how to do it beyond XP.

4.2 A Look at PE Relocations

PE relocations have been known to harbor all sorts of weirdness. For example, some MIPS-specific types were supported on x86, Sparc or Alpha. One type appeared and disappeared in Windows 2000.

Typically, PE relocations are limited to a simple role: whenever a binary needs to be relocated, the standard Type 3 (`HIGH_LOW`) relocations are applied by adding the delta $LoadedImageBase - HeaderImageBase$ to each 32 bit immediate.

However, more relocation types are available, and a few of them present interesting behavioral differences between operating system releases that we can use.

Type 9 This one has a very complicated 64-bit formula under Windows 7 (see Roy G Big's `vcode2.txt` from Valhalla Issue 3 at `http://spth.virii.lu/v3/`), while it only modifies 32 bits under XP. Sadly, it's not supported anymore under Windows 8. It is mapped to `MIPS_JMPADDR16`, `IA64_IMM64` and `MACHINE_SPECIFIC_9`.

Type 4 This type is the only one that takes a parameter, which is ignored under versions older than Windows 8. It is mapped to `HIGH_ADJ`.

Type 10 This type is supported by all versions of Windows, but it will still help us. It is mapped to `DIR64`.

So Type 9 relocations are interpreted differently by Windows XP and 7, but they have no effect under Windows 8. On the other hand, Type 4 relocations behave specially under Windows 8. In particular, we can use the Type 4 to turn an unsupported Type 9 into a supported Type 10 only in Windows 8. This is possible because relocations are applied directly in memory, where they can freely modify the subsequent relocation entries!

[4] `http://www.phreedom.org/research/tinype/`
[5] Fast Small Good, by bart/xt
[6] See "TLS AddressOfIndex in an Imports descriptor" for differentiating OS versions by use of Corkami's `tls_aoiOSDET.asm`.

4.3 Implementation

Here's our plan:

1. Give a user-mode PE a kernel-mode `ImageBase`, to force relocations,
2. Add standard relocations for code,
3. Apply a relocation of Type 4 to a subsequent Type 9 relocation entry:
   - Under XP or Win7, the Type 9 relocation will keep its type, with an offset of $0f00h$.
   - Under Win8, the type will be changed to a supported Type 10, and the offset will be changed to $0000h$.
4. We end up with a memory location, that is either:

   XP Modified on 32b ($00004000h$),

   Win7 modified on 64b ($08004000h$), or

   Win8 left unmodifed ($00000000h$), because a completely different location was modified by a Type 10 relocation.

```
;relocation Type 4, to patch unsupported relocation Type~9 (Windows~8)
block_start1:
    .VirtualAddress dd relocbase - IMAGEBASE
    .SizeOfBlock dd BASE_RELOC_SIZE_OF_BLOCK1

    ; offset +1 to modify the Type, parameter set to -1
    dw (IMAGE_REL_BASED_HIGHADJ << 12) | (reloc4 + 1 - relocbase), -1
BASE_RELOC_SIZE_OF_BLOCK1 equ - block_start1

; our Type 9 / Type 10 relocation block:
; Type 10 under Windows8,
; Type 9 under XP/W7, where it behaves differently
block_start2:
    .VirtualAddress dd relocbase - IMAGEBASE
    .SizeOfBlock dd BASE_RELOC_SIZE_OF_BLOCK2

; 9d00h will turn into 9f00h or a000h
reloc4 dw (IMAGE_REL_BASED_MIPS_JMPADDR16 << 12) | 0d00h
BASE_RELOC_SIZE_OF_BLOCK2 equ $ - block_start2
```

We now have a memory location modified transparently by the loader, with a different value depending on the OS version. This can be extended to generate different code, but that is left as an exercise for the reader.

5 This ZIP is also a PDF

by Julia Wolf

We the editors have lost touch with the author, who submitted the following napkin sketches in lieu of the traditional LaTeX or ASCII prose. Please note when forming your own submissions that we do not accept napkins, except when they are from Julia Wolf or from John McAfee.

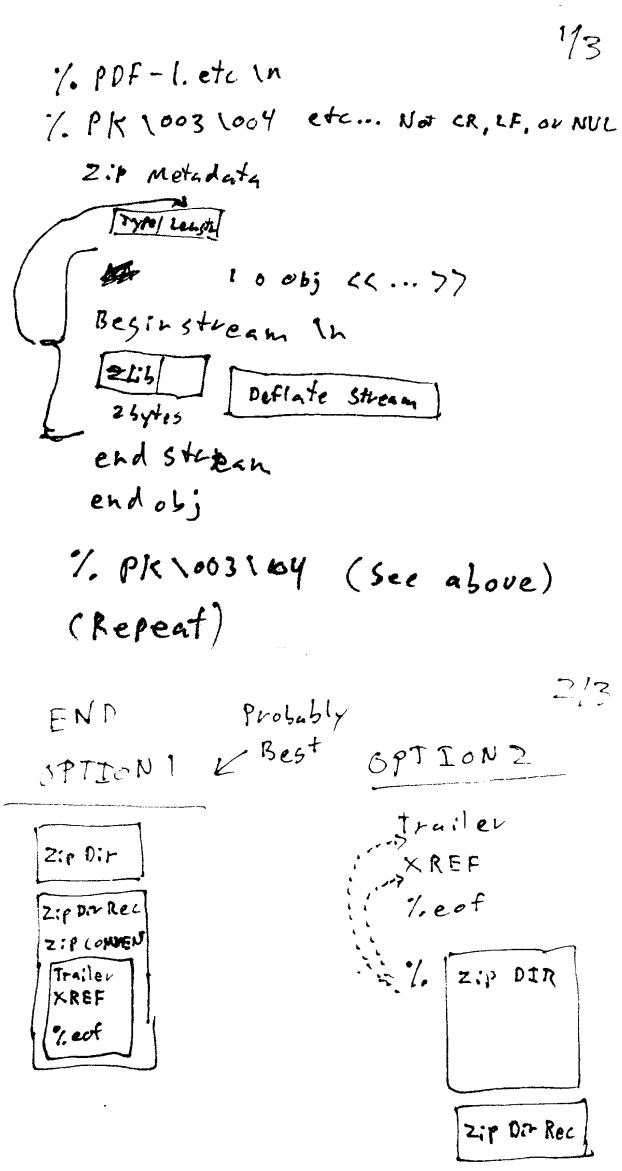
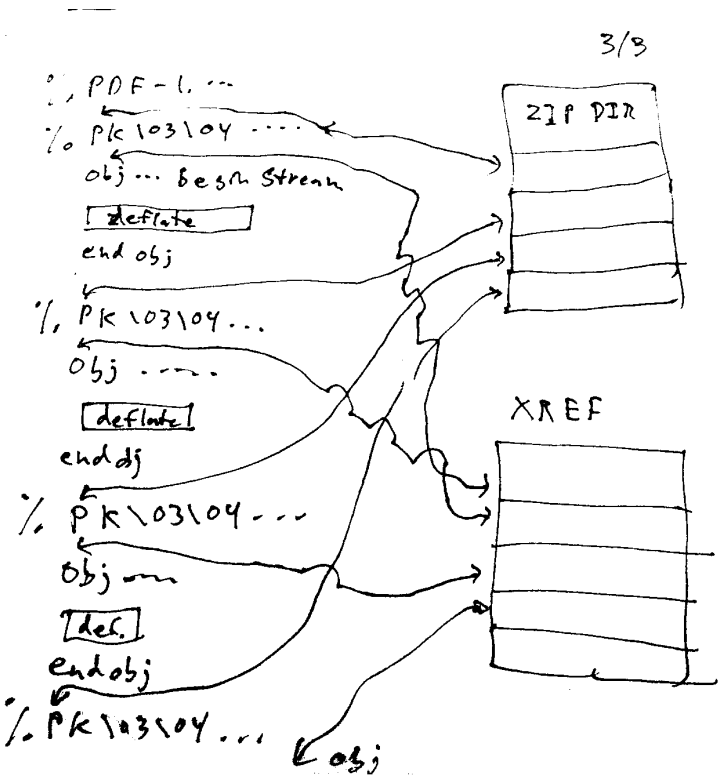


Figure 2: Napkins 1 and 2



4/3

cat foo.pdf bar.zip > buz.pdf

zip -A buz.pdf

or ...

cat foo.pdf bar.zip foo.pdf > buz.pdf

zip -A buz.pdf

Or ...

echo -e "trailer << /root /.... >> \n xref \n ....
etc. %-EOF" > comment.txt

cat foo.pdf bar.zip > bbz.pdf

zip -z comment.txt ~~bar.zip~~ ~~-A buz.pdf~~ Stuff

zip -A buz.pdf

Figure 3: Napkins 3 and 4

6 Burning a Phone

by Josh "@m0nk" Thomas

Earlier this year, I spent a couple months exploring exactly how power routing and battery charging work in Android phones for the DARPA Cyber Fast Track program. I wanted to see if I could physically break phones beyond repair using nothing more than simple software tricks and I also wanted to share the path to my outcomes with the community. I'm sure I will talk at some point about the entire project and its specific targets, but tonight I want to simply walk through breaking a phone, see what it learns us and maybe spur some interesting follow on work in the process.

Because it's my personal happy place, our excursion into kinetic breakage will be contained to the pseudo Linux kernel that runs in all Android devices. More importantly, we will focus the arch/arm/mach-msm subsystem and direct our curiosity towards breaking the commonplace NAND Flash and SD Card hardware components. A neighbor specifically directed me not to include background information in this write-up, but we have to start somewhere prior to frying and disabling hardware internals and in my mind the logical starting point is the common power regulation framework.

The Linux power regulation framework is surprisingly well documented, so I will simply point a curious reader to the kernel's documentation at Documentation/power/regulator/overview.txt. For the purpose of breaking devices, all we really need to understand at the onset are these three things.

- The framework defines voltage parameters for specific hardware connected to the PCB.
- The framework regulates PMIC and other control devices to ensure specific hardware is given the correct voltages.
- The framework directly interacts with both the kernel and the physical PCB, as one would expect from a (meta) driver

It's also worth noting that the PCB has some (albeit surprisingly limited) hardwired protections against voltage manipulations. Further, the kernel has a fairly robust framework to detect thermal issues and controls to shut down the system when temperature thresholds are exceeded.

So, in essence, we have a system with a collection of logical rules that keep the device safe. This makes sense.

Glancing back at our target for attack, we should quickly consider end result potentials. Do we want to simply over volt the NAND chip to the point of frying all the data or do we want something a little more subtle? To me, subtle is sexy..., so let's walk though simply trying to ensure that any NAND writes or reads corrupt any data in transit or storage.

On the Sony Xperia Z platform, all NAND Flash and all SD-Card interactions are actually controlled by the Qualcomm MSM 7X00A SDCC hardware. Given we RTFM'd the docs above, we simply need to implement a slight patch to the kernel:

```
project kernel/sony/apq8064/
diff --git a/arch/arm/mach-msm/board-sony_yuga-regulator.c
          b/arch/arm/mach-msm/board-sony_yuga-regulator.c

-- RPM_LDO(L5, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L5, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),

-- RPM_LDO(L6, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L6, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),
```

Wow that was oddly easy, we simply upped the voltage supplied to the 7X00A from 2.95V to 5.9V. What did it do? Well, given this specific hardware is unprotected from manipulation across the power band at the PCB layer and at the internal silicon layer, we just ensured that all voltage pushed to the NAND or

SD-Card during read / write operations is well above the defined specification. The internal battery can't actually deliver 5.9V, but the PMIC we just talked to will sure as hell try and our end result is a NAND Flash chip that corrupts nearly every block of storage it attempts to write or read. Sometimes the data comes back from a read request normal, but most of the time it is corrupted beyond recognition. Our writes simply corrupt the data in transit and in some cases bleed over and corrupt neighbor data on storage.

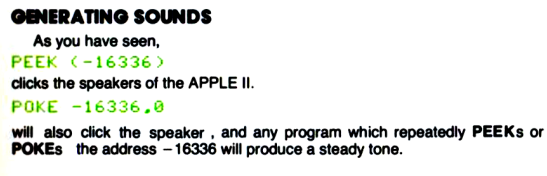
Overall, with two small values changed in the code base of the kernel we have ensured that all persistent data is basically unusable and untrustworthy. Given the PMIC devices on the phone retain the last valid setting they've used, even rebooting the device doesn't fix this problem. Rather, it actually makes it much worse by corrupting large swaths of the resident codebase on disk during the read operation. Simply, we just bricked a phone and corrupted all data storage beyond repair or recovery.

If instead of permanently breaking the embedded storage hardware we wanted to force the NAND to hold all resident data unscathed and ensure that the system could not boot or clean itself, we simply need to under-volt the controller instead of upping the values.

If you find this interesting, look forward to my release of a longer variant of this technique that targets all hardware soldered in the phone PCB in paper form on github soon.

7 A Sermon concerning the Divinity of Languages; or, Dijkstra considered Racist

an epistle from the Rt. Rvd. Pastor Manul Laphroaig,
for the Beloved Congregation of the First United Church of the Weird Machines.


GENERATING SOUNDS

As you have seen,

```
PEEK (-16336)
```

clicks the speakers of the APPLE II.

```
POKE -16336,0
```

will also click the speaker , and any program which repeatedly **PEEK**s or **POKE**s the address −16336 will produce a steady tone.

Figure 4: Excerpt from Apple][Basic Programming (1978)

Indulging in some of The Pastor's Finest, I proclaim to my congregation that there is divinity in every programming language.

"But," they ask, "if there is divinity in all languages, where is the divinity in PHP? Though advertised as a language for beginners, it is impossible for even an expert to code in it securely."

Pouring myself another, I say, "PHP teaches us that memory-safe string concatenation is just as dangerous as any stupid thing a beginner might do in C, but a hell of a lot easier to exploit. My point is not in that PHP is so easy to write, as it isn't easy to write safely; rather, the divinity of PHP is in that it is so easy to exploit! Verily I tell you, dozens of neighbors who later learned to write good exploits first learned that one program could attack another by ripping off SQL databases through poorly written PHP code.

"If a language like PHP introduces so many people to pwnage, then that is its divinity. It provides a first step for children to learn how program execution goes astray, with control and data so easy to mangle."

"But," they ask, "if there is divinity in all languages, where is the divinity in BASIC? Surely we can mock that hellish language. Its line numbers are ugly, and the gods themselves laugh at how it looks like spaghetti."

Pouring myself another, I proclaim, "The gods do enjoy a good laugh, but not at the expense of BASIC! While PHP is aimed at college brogrammers, BASIC is aimed at children. Now let's think this through carefully, without jumping to premature conclusions.

"BASIC provides a learning curve like a cardboard box, in that when trapped insider a clever child will quickly learn to break out. In the first chapter of a BASIC book, you will find the standard Hello World.

```
10 PRINT "Hello World"
```

"Groan if you must, but stick with me on this. In the sixth chapter, you will find something like the following gem.

```
250 REM This cancels ONERR in APPLE DOS
260 POKE 216, 0
```

"Sit and marvel," I say, "at how dense a lesson those two lines are. They are telling a child to poke his finger into the brain of the operating system, in order to clear an APPLE DOS disk error. How can C or Haskell or Perl or Python begin to compete with such educational talent? How advanced must you be in learning those languages to rip a constant out of the operating system's brain, like PEEK(222) to read the error status or POKE 216, 0 to clear it?"

A student then asks, "But the code is so disorganized! Professor Dijkstra says that all code should be properly organized, that GOTO is harmful and that BASIC corrupts the youth."

Pouring myself another, I say "Dijkstra's advice goes well enough if you wish to program software. It is true that BASIC is a horrid language for writing complex software, but consider again the educational value in spaghetti code.

"Dijkstra says that a mind exposed to BASIC can never become a good programmer. While I trust his opinions on algorithms, his thoughts on BASIC are racist horse shit.

"A mind which has **not** been exposed to BASIC will only with great difficulty become a reverse engineer. What does a neighbor who grew up on BASIC spaghetti code think when he first reads unannotated disassembly? As surely as the gostak distims the doshes, he knows that he's seen worse spaghetti code and this won't be much of a challenge!

"Truly, I am in as much awe of the educational genius of BASIC as I am in awe of the incompetence of the pedagogues who lock children in a room with a literate adult for a decade, finding those children to still be unable or unwilling to read at the end. Lock a child in a room with an APPLE][and a book on BASIC, and in short order a reverse engineer will emerge.

"There is divinity in all languages, but BASIC might very will be the most important for teaching our profession."

"But," they ask, "if there is divinity in all languages, where is the divinity in Java?"

Pouring myself another, I drink it slowly. "The lesson is over for today."

8 A Call for PoC

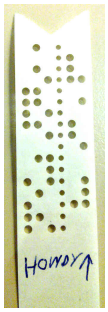
by Rt. Revd. Preacherman Pastor Manul Laphroaig

We stand, sit, or simply relax and chill on the shoulders of the giants, *Phrack* and *Uninformed*. They pushed the state-of-the-art forward mightily with awesome, deep papers and at times even with poetry to match. And when a single step carries you forward by a measure of academic years, it's OK to move slowly.

But for the rest of us dwarves, running around or lounging on those broad shoulders can be so much fun! A hot PoC is fun to toss to a neighbor, and who knows what some neighbor will cook up with it for the shared roast of the vuln-beast? A neighbor might think, "my PoC is unexploitable" or "it is too simple," but verily I tell you, one neighbor's PoC is the missing cog for another neighbor's 0day. How much PoC is hoarded and lies idle while its matching piece of PoC wastes away in another hoard? Let's find out!

8.1 Author guidelines

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 exploit Dan's random number generator; teach me how to make a cartoon that prints differently each time by abusing the printer's postscript interpreter; or, teach me how to do system calls in Cisco shellcode. Don't tell me that it's possible; teach me how to do it myself.

Like an email, I expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for later drafts. 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.

8.2 Other Departments

Editor at Large	Rt. Revd. Preacherman Pastor M.L.
Dept. of Bringing APT Home	Cultural attaché of the 41st Directorate
Dept. of Fail	FX of Phenoelit
Ethics Board	The Grugq
Dept. of Busting BS	pipacs
Poet Laureate	Ben Nagy
Dept. of Rejections	Academic Refugee
Dept. of Drama	Xbf
Dept. of PHY	Michael Ossmann