Children’s Bible Coloring Book of PoC || GTFO
Issue 0x02, an Epistle to the 30th CCC Congress in Hamburg

Composed by the Rt. Revd. Pastor Manul Laphroaig to put pwnage before politics. pastor@phrack.org

December 28, 2013

Legal Note: If you have received this book without a cover or crayons, you should be aware that your friends are awesome! It was produced by samizdat from the freely available pocorgtfo02.pdf. Neighbor, you have our blessing to copy this as you like. Yodel it, preach it, doodle it, and share this gospel with the whole of creation, ’cause we don’t give a shit.

1 Call to Worship

Please join me in reading this third 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 two 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 or the second in São Paulo.

This edition is written to the fine neighbors of the Chaos Computer Club in honor of their thirtieth congress, to be held this December in Hamburg. As in prior issues, you’ll find plenty of pwnage, some neighborly preaching, and no politics.

In Section 2, Pastor Laphroaig preaches that in the tradition of Noah and of Howard Hughes, we should build our own fucking birdfeeders.

Brother Myron Aub takes a break from his evangelical promotion of Graphitics to teach us a little about the PGP Message format in Section 3. It turns out that RFC 4880 gives him just enough room to encode an LZ-compression quine within a message, and the PGP interpreter is just “smart" enough to keep decoding it ’till the cows come home. Perhaps other weird machines remain to be found?

Natalie Silvanovich shares in Section 4 her techniques for reliably dropping shellcode into the Tamagotchi’s 6502 controller from malicious plugin cartridges. Her exploit requires a number of nifty tricks, not least of which is that the some bits of the program counter are ignored in this architecture, so her victim executes the right code from the wrong address! It is feared that this technology might be used

1Because things marketed as “smart” usually aren’t, at least not for the buyer’s benefit. Truly, the world does occasionally need reminding that stupid is as stupid does.
by the Royal Canadian Mounted Police to fuel a Cyber War of 1812 against the State of New Hampshire and the People's Republic of Vermont. Both American and Canadian neighbors can rest assured that this one would have the same winner as the original, Non-Cyber War of 1812.

Travis Goodspeed shares a grab-bag of tricks for exploiting microcontrollers in Section 5. Learn how to combine a Write and a Checksum primitive with weirder properties of Flash memory into a bitwise Read primitive when exploiting microcontrollers, how to NOP-out instructions without erasing Flash pages, and how to use bootloader ROMs for a return-to-libc attack.

Bx Shapiro had a nifty article in PoC∥GTFO 0:5 in which she showed out to return from ELF to libc. That article ended with a challenge to our readers, asking you fine folks to figure out how in living hell parameters could be passed to the function begging called. In Section 6, she rises to her own challenge, showing you how to call putchar() from an ELF Weird Machine without having any of your own native code.

Dave Weinstein in Section 7 explains why POKE 62975, 0 will brick a Trash 80 Model 100 until that poor machine is put out its misery by a cold reset. Feel free to try it out in your emulator and consider that many Automatic Exploit Generators aren't very good at predicting the effects of a write-once-anywhere vuln.

Ange Albertini explains the internal organization of this issue's PDF in Section 8. Curious readers might want to run qemu-system-i386 -fda pocorgtfo02.pdf in order to experience all the neighborliness that this issue has to offer.

In PoC∥GTFO 01:02, Dan Kaminsky shared with us a 4-line RNG for Javascript, challenging our readers to exploit it. It had no whitening, no scrambling, and no other defenses, so any weakness in the principle ought to have been exploitable. In proper PoC∥GTFO fashion, Joernchen demonstrates such a vulnerability in Section 9, by observing that some versions of Firefox bias toward producing bytes of low Hamming weight.

Section 10 contains Ben Nagy's latest masterpiece, sure to get you, dear reader, on all sorts of watchlists. We half-heartedly apologize in advance to any of our readers at spooky agencies who have to explain having this magazine to their employers.

Finally, in Section 11, we do what churches are best at and pass the collection plate. Please consider giving alms of 0day and PoC to those who are poor in spirit.

Artwork in this issue was created by Ra of Tama-Zone, Stefan Bauwens, and others. The painting featured in the museum on page 31 is in remembrance of the one first drawn by Mirromaru in red creeper cards at the 29th Congress, then quickly censored due to controversy.

We the editors are aware that some of the illustrations might be offensive to our more sensitive readers, either for reasons of vulgarity or blasphemy. In both cases, we rely on the Bill Hicks Defense. "Buddy, we're Christians, and we don't like what you said."

"So forgive me!"
2  A Parable on the Importance of Tools; or,  
Build your own fucking birdfeeder.

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

Grace and Peace to you!

Once there was a wine-maker named Noah, the sort of fella you’d be happy to share a beer with. He made damned good wine, but one day he started building a boat.

“Why are you building that?” they’d ask, “Are the voices in your head telling you that it’s gonna rain?”

“Nope,” he’d say, “Just toolin’ around.”

They showed him yacht catalogs and boating magazines. “Look, man, you can just buy one at the store.”

“Haven’t got the money,” he’d say and then get back to building the frame or bending boards for the hull.

“Well, you could afford to rent a boat for the weekend.”

Now Noah was a patient guy, but everyone has his limit. “I’m building my own fucking birdfeed,” he’d say, “because they’ve got wood at the store.”

And there was a fella named Howard Hughes, a crazy old millionaire. Back in the thirties, he built his own air force to film a movie about the first World War, so during the forties, when Roosevelt needed an air force of his own, he bought Howie’s.

Howie Hughes built other birdfeeders. He made the H4 Hercules, the world’s largest airplane and a damned big boat, out of wood. It was five stories tall with a hundred meter wingspan. First flying in 1947, nothing approaching its size was seen for another forty years.

During the cold war, when the CIA wanted to recover a sunken Soviet submarine, K-129, they called ol’ Howie up. “Howie,” they said, “We’ve gotta keep this real quiet. Don’t tell anyone.”

So the next day, Howard Hughes held a press conference! “There are giant blobs of copper on the ocean floor,” he lied, “and I’m building a big-ass boat with a big-ass crane to pick them up and drop them on the deck. It’ll be so efficient that I’ll put the other copper mines out of business.”

So while folks were scrambling to invest in his copper company and divest from the real ones, Howie built the Hughes Glomar Explorer. True to his word it was a big-ass boat with a big-ass crane, but instead of picking up copper blobs it lifted that submarine off the ocean floor and dropped it on the deck.

How could he do these things? Because he built his own fucking birdfeeders, that’s how.

So when you’re tooling around with a from-scratch tool, your own hex editor or interactive disassembler, and your neighbors tell you to use 010 or to use IDA or to use this or use that, do what Noah and Howie would do. Look ’em in the eye and say,

“I’m building my own fucking birdfeeder.”
Pastor Laphroaig tells us that when the streams of our computation are unclear, it’s often because the SEO Experts are enjoying their goats upstream.

Pastor Laphroaig says to the SEO Experts, “Not with my flock!”
3 A PGP Matryoshka Doll

by Brother Myron Aub

Take out your favourite matryoshka doll, neighbour. Now piece by piece, open it until you can open it no longer. Every piece is smaller and closer to the end of the experience, and then—it stops: you can open the smallest piece no more.

But beware, neighbour! Not all matryoshka dolls behave like this. Some matryoshka craftsmen are tempted by the devil’s lures. They see no farther than the devil’s unholy promises of extensibility and compactness when they craft a matryoshka doll that can compress a larger one to fit within it! And our good neighbour Phil Zimmerman fell prey to this lure when designing the PGP doll format.2

When you want to send a message, you must first stuff it into a literal doll. You can then enclose that in an encrypted doll, a signed doll, or a compressed doll. How do you assemble these together? However you please! You can put your literal doll inside a signed doll inside an encrypted doll inside a compressed doll. Naturally, ciphertext compresses poorly, so this would be a stupid way to nest a PGP matryoshka doll. Normally you put your literal doll inside a signed doll inside a compressed doll inside an encrypted doll, but you can do it stupidly if you like.

And how do you open a PGP matryoshka doll? Since the sender could have assembled it however they pleased, you must be ready for anything. If you see an encrypted doll, you decrypt it and open the enclosed smaller doll. If you see a signed doll, you verify its signature—throwing it away if it fails to verify—and open the enclosed smaller doll. If you see a literal doll, you’re done and you read the message.

But what if you get a compressed doll? You decompress it—and hope there are no vulnerabilities in your system’s zlib—but unless some idiot tried to compress ciphertext, the enclosed doll will be bigger than the doll you just opened.

‘Surely,’ you say, ‘if someone assembled a PGP doll for me, it must have a literal doll buried inside it!’ But no, my poor, naïve neighbour! There is no rule that all PGP dolls be assembled like that. With the help of our neighbourly neighbour Russ Cox,3 and with a dab of holy water to dispel the devil’s temptations to misuse this black magic, we can craft a voodoo PGP doll from a quine, a self-reproducing program written in the Lempel-Ziv compression language, that bites any who naïvely try to open it up.

Our neighbour Tavis Ormandy discovered similar unholiness in IPsec.4 What other matryoshka dolls can you turn into voodoo dolls, good neighbour?

---

2RFC 4880, ‘OpenPGP Message Format’
3Russ Cox, ‘Zip Files All the Way Down’, 2010-03-18
4Tavis Ormandy, ‘BSD derived RFC 3173 IPcomp encapsulation will expand arbitrarily nested payload’, CVE-2011-1547, posted to full-disclosure 2011-04-01
Hey kids! Can you reverse engineer this shellcode from the picture?
Tamagotchis are an excellent target for reverse engineering for a number of reasons: They have a limited number of inputs and outputs, they run on a poorly documented 6502 microcontroller and they’re, well, Tamagotchis. Recently, I discovered a technique for reliably executing foreign code on a Tamagotchi.

Let’s begin at the beginning. Modern Tamagotchis run on a GeneralPlus GPLB52X LCD controller, a lightweight 6502 controller that uses an internal mask ROM for all code and some data. This means that exploitation is necessary to free the Tamagotchi from the shackles of its read-only code. Also, in the absence of any debug outputs, code execution provides valuable insight into the internals of the Tamagotchi and its MCU.

There are four inputs into a Tamagotchi that can be manipulated by the user. (1) The buttons, (2) the EEPROM that saves the Tamagotchi state across resets, (3) the IR interface and (4) certain accessories containing external SPI memory called figures. Attempts to find useful bugs in the EEPROM and IR interface were unsuccessful, so I moved onto the figures. Eventually I found an exploitable bug in how the Tamagotchi processes figure data.

When attached to a Tamagotchi, figures add extra functionality, such as games or items. So attaching a figure might allow your Tamagotchi to play shuffleboard, purchase a vacuum cleaner or attend 30c3. The bug I found was in the processing of game data. Game logic is not actually included in the figure data; rather, the figure provides an index to the game logic in the Tamagotchi’s mask ROM. Changing this index causes some very strange behavior. If the index is an expected value, from 0 to about 0x20, a game will be played as expected, but for higher indexes, the device will freeze, requiring a reset. Even stranger, if the index is very high (0xD8 or higher), the Tamagotchi jumps to a different, valid screen, such as feeding the Tamagotchi or giving it a bath, and the Tamagotchi functions normally afterwards. This made me suspect that the game index was used as an index into a jump table and that freezing was due to jumping to an invalid location.

With no way to gain additional information about the cause of the behavior, and about 200 possible vulnerabilities, it made sense to fill up as much memory as possible with a NOP sled, try all possible indexes, and hope that one caused a jump to the right location. Unfortunately, the only memory controllable by the figure is the LCD RAM, so I filled that with NOPs and shellcode. (The screen data starts at 0x1C80 in the figure memory, and maps to 0x1000 in the Tamagotchi memory, for people trying this at home.) After several tries and some fiddling the shellcode, index 0xD4 lead to very unreliable code execution. This code execution allowed me to perform a complete ROM dump of the Tamagotchi, which in turn led to the ability to better analyze the bug.

The following code contains the vulnerability. Please note that the current state (current_state_22) is set from the game index without validation.

```
seg004:4E2E  LDA   byte_1A4
seg004:4E31  BEQ   loc_44E39
seg004:4E33  LDA   gameindex2
seg004:4E36  JMP   loc_44E3C
seg004:4E39  LDA   gameindex1
seg004:4E3C  CLC
seg004:4E3D  ADC  #$27
seg004:4E3F  STA   current_state_22
seg004:4E41  JMP   locret_44E4C
```

5The important index is located at address 0x18 in figure memory.
The main Tamagotchi execution loop checks the state based on a timer interrupt, then makes a state transition if the state has changed. The state transition is as follows.

```
ROM: EFE8  LDX current_state_22
ROM: EFEA  LDA $F00E,X
ROM: EFF0  STA change_page
ROM: EFF2  BEQ loc_F001
ROM: EFF4  LDA #0
ROM: EFF6  LDA #$40 ; '@'
ROM: EFF8  STA current_page
```

In essence, the Tamagotchi looks up the page of the state in a table at 0xF00E, then jumps to address 0x4000 in that page. Looking at this code, it is clear why my first exploit was unreliable. $D4 + 0xF00E + 0x27 is 0xF109, which resolves to a value of 0x3c. Since the Tamagotchi only has 19 pages, this is an invalid page number. Testing what would happen if the MCU was provided an invalid page, addresses 0x4000 and up resolved to 0xFF.

This means that there are two possibilities of how this exploit works. Either the memory addresses are floating and sometimes end up with values that, when executed, send the instruction pointer to the LCD RAM, or the undefined instruction 0xFF, when executed, puts the instruction pointer into the right place, sometimes. Barring bizarreness beyond my wildest imagination, neither of these possibilities would allow for the exploit to be made more reliable though manipulation of the figure data.

Instead, I looked for a better index to use, which turned out to be $CD. $CD + 0xF00E + 0x27 is 0xF102, which maps to part of the LCD segment table, which has a value of 4. Jumping to 0x4000 in page 4 immediately indexes into another page table.

```
seg004:4000  LDA #$D
seg004:4002  STA $34
seg004:4004  LDA #$40 ; '@'
seg004:4006  STA $35
seg004:4008  LDA $22
seg004:400A  JMP jump_into_table_D27F
```

This index is also out of range, and indexes into a code section:

```
seg004:41F5  INC $11E
```

Interpreted as a pointer, however, this value is 0x1EEE. The LCD RAM range is from 0x1000 to 0x1200, but fortunately, bits 2-7 of the upper byte of addresses in the 0x1000-0x2000 range are ignored, so reading 0x1EEE returns the value at 0x10EE. This means that playing a game with the index of 0xCD will execute code in the LCD RAM every time!

While reading POC\|GTFO obligates you to share a copy with a neighbour, trying this on your own Tamagotchi is only strongly recommended. Further instructions can be found by unzipping the PDF of this issue.
"The ancient teachers of this science promised impossibilities and performed nothing. The modern masters promise very little; they know that metals cannot be transmuted and that the elixir of life is a chimera but these philosophers, whose hands seem only made to dabble in dirt, and their eyes to pore over the microscope or crucible, have indeed performed miracles. They penetrate into the recesses of nature and show how she works in her hiding-places. They ascend into the heavens; they have discovered how the blood circulates, and the nature of the air we breathe. They have acquired new and almost unlimited powers; they can command the thunders of heaven, mimic the earthquake, and even mock the invisible world with its own shadows." – Shelley 3:16
5 Some Shellcode Tips for MSP430 and Related MCUs

by Travis Goodspeed

Howdy y’all,

I’m writing this to introduce you as an exploiter of desktops and servers to some of the tricks that I’ve used in writing shellcode for microcontrollers, with examples from the MSP430 in particular. You can try most of these examples on a GoodFET or Facedancer board, and many of them are portable to other embedded targets, such as AVR or the lower-end ARM devices.

5.1 Flash Patching is Weird

In Unix and Windows, you are used to processes operating within virtual memory. On a microcontroller, they often run directly in physical memory, so the rules are rather different. It helps to take the German approach, learning all of the rules to get away with things that ought to be illegal.

The first difference you’ll run into on the MSP430 is that code runs in-place from Flash memory. Flash has some very different rules from RAM, because it’s a different technology and a proper programmer knows better than to rely on layers of abstraction.

- Flash is erased to ones as segments or globally, never as bytes or words.
- Flash writes clear bits at word granularity, but can’t set them.
- Flash writes require a safety password to be written into a register.

Thus, to do a normal write to Flash, an MCU programmer is taught to first disable the Flash write protection and configure the right special-function registers, then erase the entire page, then rewrite the entire page. Many programmers never bother, opting for an external memory chip or relying on battery-backed RAM.

To make smaller changes, there’s another option. After disabling Flash, a neighbor could clear individual bits rather than rewriting the entire page. This is handy for regular developers to do what’s called EEPROM Emulation, which emulates memory that can be written byte-wise, but it’s also damned useful when patching code in-place.

For example, Figures 1 and 2 show that 0x3Cxx is an unconditional Jump while 0x38xx is a conditional Jump if Less Than instruction. If we overwrite a JMP instruction with 0x3BFF, it will have the effect of bitwise ANDing that instruction with 0x3BFF, changing the 3C opcode to a 38 while retaining the jump offset.
Since MSP430 instructions are 16-bit word aligned, the 10-bit PC offset is multiplied by two and then added to the program counter. 0x3FFF is an unconditional jump backward by one word, or an unconditional infinite while loop. If you zero-out the offset by overwriting the instruction with 0x3C00, you can turn any jump instruction into a NOP.

When attacking a poorly protected bootloader, you might find yourself with the ability to write and to checksum, but not to read. If you can write without erasing, then writing all 1’s with a single 0 will change the checksum if and only if that bit previously was a 1. Repeating for each bit of Flash is slow, but it might get you a firmware dump.

5.2 Efficient Shellcode

Quite often, the first thing you’ll do with shellcode is to dump out the state of the microcontroller being attacked. It’s worth studying ways to make that code in as few bytes as possible, as a microcontroller generally processes very small packets and you won’t have room for anything fancy.

To quickly dump memory on an architecture that you don’t know very well, it helps to have simple code that already has its environment configured. The code should be completely oblivious to timing, and it should access as few structures as possible. It should also be portable, requiring neither knowledge of its position in memory nor knowledge of the specifics of the rest of the device motherboard at compile time.

My solution is to blink the LEDs, half with a clock and half with data, to dump all of the memory to an SPI sniffer. The LEDs that light up with consistent brightness are the clock, while those that sporadically become very bright or very dim are the data. Tapping one of each with my handy Saleae Logic analyzer gives me a firmware dump.

5.3 Mask ROMs have Useful Gadgets

In my WOOT ‘09 paper with Aurélien Francillon, we toyed around with using the MSP430’s BSL (BootStrap Loader) ROM to aid in exploiting an unknown executable. That paper concerns exploiting firmware without having a copy, but I’ll recount one of its tricks here.

The MSP430 BSL has two entry points. The first is the Hard Entry Point, whose address is always stored at 0x0C00. By twiddling the reset and test pins with proper timing, the chip will boot from this address instead of from the RESET handler in the interrupt table.

The second entry point is called the Soft Entry Point, and it is rather poorly documented. The original idea was that a program could return into the bootloader ROM by branching to the address stored at 0x0C02, with some of the initialization routines skipped. One of these routines is the instruction that initializes the register holding password protection, so by setting or clearing a bit in that register, the calling application can enable or disable password checking.

While the soft entry point is sometimes useful to an MSP430 developer, it’s damned useful for an attacker. On an MSP430F1612, my favorite shellcode for dumping firmware is a bit like the following, which assembles to just six bytes of memory.

```
mov #0xFFFF, r11 ;; Disable BSL password protection.
br &0x0c02 ;; Branch to the BSL Soft Entry Point
```
5.4 Unused RAM is Not Erased at Reboot

In larger machines, memory which is not used by a process is not mapped into that process’s virtual memory. In microcontrollers, it is still accessible, since the code is running with physical rather than virtual memory. Rather than reset every RAM word during a reboot, most microcontrollers simply leave it alone and let the program take care of clearing its values.

Now an MSP430 application is compiled with a view of memory that it sparingly uses. GCC, for example, will allocate code (.text) into Flash from the lowest Flash address in its linker script.

RAM is only used by the compiler for data, never for code, unless the linker script is carefully and intentionally hand-crafted. It is divided into two segments by the linker, .data and .bss. The .data region is initialized by copying the data over from Flash, while the .bss region is initialized to zero through a simple while() loop. This provides us with two nifty tricks.

The first trick is that, given a poor POKE gadget, we can slowly place a large chunk of shellcode into upper regions of RAM. For example, an MSP430F2618 has enough RAM to fit the GoodFET firmware, so a device using that chip could have the GoodFET firmware itself act as second-stage shellcode! Smaller chips, such as the MSP430F2274, could have a Flash driver loaded into unused RAM, with third-stage shellcode written into unused Flash.

5.5 Where Flash is Protected, RAM is Not

Recalling that unused RAM is never cleared by an application, let’s abuse that behavior in a second way.

Back in 2010, Texas Instruments released their ZStack implementation of Zigbee for use with the Smart Energy Profile. I found that the random number generator was crap, and they patched that bug. So how was little ol’ me supposed to get more Zigbee Smart Energy Profile keys without a Certicom license?

The remaining vulnerability was a combination of the BSL ROM with the ZStack firmware. ZStack relied upon the BSL ROM and the JTAG fuses to prevent keys and firmware from being read out of the device, but the BSL ROM was only intended to keep code from being read out of the device. A second bug in that Zigbee stack was that keys were stored in the .data segment instead of the .text segment, so the firmware would copy the key from Flash into RAM during startup.

As a quick recap, the bootloader requires a password to run most commands, but some are unprotected. Among them are the ones to supply a password and the Mass Erase command, which wipes all of Flash and resets the password, which is stored in Flash, to 32 bytes of 0xFF.

So to get keys out of locked ZStack devices, I just needed to use the serial bootloader, first sending the command to Mass Erase and then—without losing power—to supply a password of all 0xFF and then to dump all of RAM to disk. A little bit of RAM is overwritten by the BSL’s call stack, but only the lowest 32 bytes. Everything else is saved.

I hope you find these tricks to be handy. If you’d like to hear more, buy me a nice India Pale Ale.

— Travis
Who would remember Noah if he had just bought a boat from the store?
Build your own fucking birdfeeder.
Calling putchar() from an ELF Weird Machine.

by Rebecca .Bx Shapiro

PASTOR'S EXORDIUM. Behold the daily miracle of the loader: it takes stored dumb bytes and makes them into a new process or splices them into a running one. The Pharisees may dismiss it as mere engineering, but verily I tell you, long after their textbooks are forgotten the loader and its Phrack exegesis will shine on, for there is more wisdom gathered in its metadata structures than can be found in a dozen OS textbooks.

Yet there is more! The binary metadata structures consumed by the loader are actually a program for the loader. A weird machine devotee will readily recognize that these data drive all the actions behind the loader’s miracle; they can be thought of as executable bytecode for the loader, which can be thought of as a virtual machine. And just as assembly with all its glorious moves, adds, and calls is encoded in opcodes and offsets, ABI metadata entries are encoded in types and addends, except that they are split into symbols and relocation structures, residing in different sections of the binary but cross-referenced by their entry numbers in the respective sections.

In this follow-up to earlier work, Bx shares more nifty tricks of programming the ELF loader with relocation and symbol data as weird assembly. This work is as advanced as it is neighborly, so please read her articles from WOOT 2013 and POC GTFO 00:05 to learn how to build a Turing-complete virtual machine out of an ELF loader and how to extend that VM to call native code. In this sermon, Bx shows us how to make system calls from ELF relocation and symbol data; full shellcode is left as an exercise to the faithful! –PML

Welcome back, friends. In the first edition of POC GTFO, I demonstrated how we can craft ELF relocation metadata to instruct the loader to make libc calls. The method I demonstrated was fairly limited and lacked the ability to do useful things such as control the arguments passed to the called function. Thus I ended the article with an unsolved challenge: How can metadata control the arguments passed to the metadata-initiated function call?

In this sermon, I will partially answer that challenge by demonstrating how to control a call to putchar() using relocation metadata.

PUTCHAR(3) bx's Programmer's Manual PUTCHAR(3)

SYNOPSIS

#include <stdio.h>

int putchar(int c);

DESCRIPTION

putchar(c) writes the character c, cast to an unsigned char, to stdout.

RETURN VALUE

putchar() returns the character written as an unsigned char cast to an int or EOF on error.

puts() and fputs() return a nonnegative number on success, or EOF on error.

One may ask “why focus on putchar()?” The answer is simple. Because putchar() is required in order to implement a full, honest-to-manual brainfuck-to-ELF metadata compiler. You may have noticed that putchar() requires only a single (byte-long) argument and have thought to yourself “I only have control over one argument!? How will that help me take over the world?” Don’t worry your pretty little

---

7How is a sermon like a binary file? Both have prescribed parts that follow each other in a conventional order, but may be skipped or used creatively by an extra neighborly preacher. Convention is there to help, but it’s the result that matters. So just think of exordium as the ELF/ABI header or vice versa and bear with the Preacher as you bear with your binary toolchain! –PML
nose off. I will provide insight on how you can control not one, not two, but three (ish) arguments to a function call!

Instead of asking how one can control the first argument to a function call, one should really be asking how can we be the last to set the RDI register (the first argument to a function as heralded by the System V amd64 ABI gospel 3:2:3, aka amd64 calling convention\(^8\)) before our metadata-driven libc function is called.

It turns out that the loader generally processes each relocation entry within a single function, although there are a few exceptions to this rule. This means that, generally speaking, the arguments that are in place during any metadata-driven function call are the arguments that were passed to the currently executing function processing the relocation entries. An exception to this “rule” occurs when relocation entries of type R_X86_64_COPY are processed. These types of relocation entries cause the loader to make a call to `memcpy()`, thus changing the values of RDI, RSI, RDX, which by convention hold the first three arguments to a function call, and in the case of a call to `memcpy(void *dest, const void *src, size_t n)` hold dest, src, and size, respectively.

Now imagine that the dynamic loader has been processing our relocation entries and now the next dynamic symbol, pointed to by the next relocation entry r0 to be processed, looks like this:

\[
\text{s0} = \{\ldots, \text{st\_value} = &\text{putchar, st\_size} = 0x0\}\]

(Note: We have already shown how to address the libc functions in past work and will not cover how to do that in this sermon. See our WOOT article and POC∥GTFO 00:05 for a thorough explanation.)

The following three relocation entries (represented here as C structs, but of course encoded in a .rel section) will make a call to `putchar()`, to print the character of our choice:

\[
\begin{align*}
\text{r0} &= \{\text{r\_offset}=<&r2->r\_addend>, \text{r\_symbol}=0, \text{r\_type}=\text{R}_X86\_64\_64, \\
&\quad \text{r\_addend}=0x0\}\;
\text{r1} &= \{\text{r\_offset}=<\text{char to print}>, \text{r\_symbol}=0, \text{r\_type}=\text{R}_X86\_64\_COPY, \\
&\quad \text{r\_addend}=0x0\}\;
\text{r2} &= \{\text{r\_offset}=&r2, \text{r\_symbol}=0, \text{r\_type}=\text{R}_X86\_64\_IRELATIVE, \\
&\quad \text{r\_addend}=&\text{putchar (filled in by r0)>}\}
\end{align*}
\]

The purpose of r0 is to write the address of `putchar()` into r2’s addend. The purpose of r1 is to setup RDI (the first argument) for r2’s function call. When it is processed, `memcpy()` is called with the following arguments: `memcpy(<char to print>, &putchar, 0)`. More generally, the call to `memcpy()` looks like: `memcpy(r1->r_offset, s0->st_value, s0->st_size)`.

After r1 is processed, 0 byes are copied from `&putchar` to `<char to print>\(^9\)`, and RDI=<`char to print`>, RSI= `&putchar`, and RDX:=0. r2, of type `R_X86_64_IRELATIVE`, instructs the loader to treat its addend as a function pointer, making a call to it(!). How’s that for a relocation-based weird assembly instruction? But, one problem: relocation entries of type IRELATIVE do not support functions that require arguments (meaning that there is no conventional way to pass them). Still, the actual function doesn’t care and will happily reach for its arguments in RDI etc.—and, luckily, we were able to set up the arguments via our relocation-entry crafted call to `memcpy()` via r1! Hence r2 will cause the loader to call `putchar()`, which will consult RDI to determine what character to print to `stdout`.

You may see the potential downfalls of manufacturing a call to `memcpy()` in order to put arguments in place for the following library call. For example, if the third argument is not zero, you need to start worrying about your first two arguments pointing to read/writable memory. However, it may be comforting to know that the value returned by the function call is written into a spot of your choosing (in `r2->r_offset`).

If you would like to further your studies of metadata-driven library calls, please refer to the elf-bf-tools repository on github.\(^{10}\) May the Great Manul keep and protect you from the Weird Machine. And let us say, amen.

---

\(^{8}\)http://www.x86-64.org/documentation/abi.pdf, pages 17-21, Fig. 3.4—and don’t ask us in what world RDI, RSI, RDX might stand for A, B, C or suchlike. This program may be brought to you by the register RDI anyhow, but let’s just say if the Manul meets the amd64 Big Bird there might be feathers flying.

\(^{9}\)Note, `memcpy` would treat it as a destination pointer, but luckily nothing gets copied here, and `memcpy` implementation isn’t paranoid about checking its arguments, since a bad pointer would trap anyway.

\(^{10}\)See syscall/putchar in https://github.com/bx/elf-bf-tools.
446 case R_X86_64_IRELATIVE:
447     value = map->l_addr + reloc->r_addend;
448     value = ((Elf64_Addr (*)(void)) value) ();
449     *reloc_addr = value;
450     break;

429 case R_X86_64_COPY:
430     if (sym == NULL)
431         /* This can happen in trace mode if an object could not be (gdb)
432          found. */
433         break;
434     memcpy (reloc_addr_arg, (void *) value,
435             MIN (sym->st_size, refsym->st_size));
436     if (__builtin_expect (sym->st_size > refsym->st_size, 0)
437         || __builtin_expect (sym->st_size < refsym->st_size, 0)
438             && GLRO(dl_verbose))
439         {
440             fmt = "
441                 %s: Symbol '%s' has different size in shared object, consider re-linking\n";
442             goto print_err;
443         }
444         break;
445     # endif

--------------
Breakpoint 6, elf_machine_rela (sym=0x601030, reloc_addr_arg=0x601241, version=<optimized out>,
    reloc=0x601318, map=0x555555773228) at ../sysdeps/x86_64/dl-machine.h:434
434     memcpy (reloc_addr_arg, (void *) value,
        (gdb) print/x *reloc
$6 = {r_offset = 0x601241, r_info = 0x5, r_addend = 0x0}
        (gdb) print refsym->st_size
$7 = 0
        (gdb) print sym->st_size
$8 = 0
        (gdb) print/x reloc_addr_arg
$9 = 0x601241
        (gdb) x/gx reloc_addr_arg
0x601241:0x0000000060103800
        (gdb) x/gx value
0x7ffff7ce1184:0x011d8b48f894153

(gdb) print/x $rsi
$5 = 0x7ffff7ce1184

(gdb) print $rdx
$10 = 0

(after memcpy)

(gdb) x/gx 0x601241
0x601241:0x000000000060103800

(gdb) print/x $rdi
$14 = 0x601241

(gdb) c
Continuing.

Breakpoint 5, elf_machine_rela (sym=0x601030, reloc_addr_arg=0x6012e8, version=<optimized out>,
reloc=0x601330, map=0x5555555773228) at ../sysdeps/x86_64/dl-machine.h:448
448 value = ((Elf64_Addr (*)(void)) value) ();

(gdb) print/x $rdi
$15 = 0x601241

(gdb) print/x value
$16 = 0x7ffff7ce1184

(gdb) x/10i value
0x7ffff7ce1184:push %rbx
0x7ffff7ce1185:mov %edi,%r8d
0x7ffff7ce1188:mov 0x313c01(%rip),%rbx # 0x7ffff7ff4d90
0x7ffff7ce118f:mov (%rbx),%eax
0x7ffff7ce1191:test $0x80,%ah
0x7ffff7ce1194:jne 0x7ffff7ce11ea
0x7ffff7ce1196:mov %fs:0x10,%r9
0x7ffff7ce119f:mov 0x88(%rbx),%rdx
0x7ffff7ce11a6:cmp 0x8(%rdx),%r9
0x7ffff7ce11aa:je 0x7ffff7ce11df

(gdb) print/x $rsi
$4 = 0x7ffff7ce1184

P.C. cards made simple—with COPYDAT!

1. Prepare the 1X artwork, using an opaque layout aid such as Chartpak, Bishop Graphics, or other similar product.
2. Make a negative: Place the artwork face down, cover with the negative material colored film side up (we recommend Scotchcal products), and expose with the Copydat. Typical exposure time is 1.5 minutes.
3. Develop the negative in developer provided with negative material.
4. Attach negative to pre-sensitized face of copper board. Place board and negative face down on Copydat. Expose. Typical exposure time: 30 seconds.
5. Save the negative for reuse, and develop the board in the developer provided.
6. Etch the board.
7. As a finishing touch, tin the board to avoid oxidation of the copper and to improve solderability.
Result: a custom, high quality, single-sided P.C. board.
With careful alignment, you can make doublesided boards too!
Alternatively, buy high-quality hardware assemblies from us — and these are predrilled as well (and feature plated-through holes):
P.S. The Copydat does a lot more than make high-quality P.C. boards. It makes superior blueinline, blackline, sepia, and other diago process copies, and you can make pressure-sensitive labels with it and even instrument front panels from pre-sensitized metal plates!!

from $149.95 (B size prints)

CELDAT Design Assoc.
P.O. Box 752
Amherst, N.H. 03031
Just as Jonah was told to preach in Nineveh,
Pastor Laphroaig was once called to preach to the harlots and tax collectors at RSA.
Asked about the experience, he said that, like Jonah,
he'd rather be thrown overboard than go back.
In his Epistle on the Divinity of Languages, PoC∥GTFO 01:07, Pastor Manul Laphroig wrote of the merits of PEEK and POKE in teaching the youth of a previous generation how to fiddle with hardware in ways the hardware did not want to be fiddled.

And so I offer to you a short example of the wonders of POKE as applied to interrupt handlers.

In 1983, Radio Shack introduced the Model 100, a copy of the Kyocera Kyotronic 85. With its 40 character wide 8-line screen, built-in 300 baud modem, and up to 32k of RAM, it was a state of the art laptop, capable of generating endless questions from passengers and crew on any flight.

In high memory, there is a vector at 0xF5FF, which allows a program to hook the keyboard/clock interrupt. Every 4 ms or so, the timer interrupt fires, and the keyboard is polled. By default, the vector is a simple RET NOP NOP.

As it happens, the very next vector in high memory is a JMP to handle the low-power situation and shut the computer down.

```
0xf5ff      0xc9 (RET)
0xf600      0x00 (NOP)
0xf601      0x00 (NOP)
0xf602      0xc3 (JMP 0x1451)
0xf603      0x31
0xf604      0x14
```

The function at 0x1431 will turn the computer off, as the code flows to the actual shutdown sequence at 0x1451:

```
0x1451      di
0x1452      in 0xba
0x1454      ori 0x10
0x1456      out 0xba
0x1458      hlt
```

Should we replace the RET at 0xF5FF (62975) with a NOP, the Model 100 will power down every time the timer interrupt fires. The only way to restore functionality is to do a cold restart of the machine, which, if I recall correctly, in this case requires removing the batteries, unplugging the machine, and disabling the internal NiCad battery. All of the contents would be lost. For those who do not know what has been done, the computer shows every sign of having simply died.

POKE 62975, 0

The only way to prevent it is to prevent access to the BASIC interpreter. Which is possible, but is a discussion for another time.
Pastor Laphroaig tells us that the news is stranger than fiction, because unlike the news, fiction requires an element of truth.
This OS is also a PDF

by Ange Albertini

A careful reader may have noticed that a bootable OS image was hidden in the last issue of PoC || GTFO, as one of the files in its dual PDF/ZIP structure (if you haven’t, download and extract it now!). This time, though, let’s hide it in plain sight. You will find by running `qemu-system-i386-fdapocorgtfo02.pdf` that the PDF file you are reading is also a bootable disk image.

8.1 Requirements

To combine two file types, we first need to list the requirements of each format and then produce a single file that meets both sets of requirements with no conflicts.

What makes a bootable disk image? An X86 machine begins booting by copying the first 512 byte sector, the Master Boot Record, into RAM and executing it. The requirements for a functional MBR are simple:

- 16 bit x86 code starts at offset 00.
- It will be executing at the 0000:7c00 address in RAM.
- It must be 512 bytes long, ending with the signature 55, AA
- Labels and primary partition tables are optional, but can go within this sector.
- It must contain code that finds and loads into RAM the code for the next boot stage (such as an OS loader).

PDF files are a mixture of text and binary fragments, which are parsed from the start of the file and delimited by words and newlines. The requirements for a valid PDF are also simple and surprisingly flexible:

- It is initially parsed as text.
- The signature “%%PDF-” must be present within the first 1024 bytes. It can be present there twice or more.
- Comment lines begin with ‘%’, which is 25 in hex.
- Binary characters other than CRLF are acceptable in a comment.
- “Multi-line” binary objects or simply larger objects can also be stored in object streams, which are declared like this:
  
  `<obj number> <revision> obj
  <\x3e\x3e
  stream
  <stream content>
  endstream
  endobj`

8.2 Strategy

In most cases, we can freely prepend anything at the start of the file as long as the above requirements are fulfilled. Luckily, the % comment character is 0x25, which encodes nicely as an x86 and instruction. Thus, the head of the file can be 25FFFF: and ax, 0xffff, which also starts a PDF comment. We can then add a jump into the next part of the code, which will be stored in a dummy object stream below, and then finish our first line. Adding a PDF signature will prevent any potential problem in case the stream object is too long: it can then contain anything, of any length, as long as it doesn’t contain the ‘endstream’ keyword.
8.3 An Unexpected Challenge

This was almost too easy, but there is a caveat to keep in mind. I’ll mention it here to save you the headache when reproducing these results.

This new challenge emerged as I was testing the bootable PDF files with different PDF readers. Since we pre-pend our MBR without altering the contents of the original document, the original’s cross-reference table XREF is no longer in sync with the actual file offsets. Technically, this makes the XREF tables corrupted.

Corrupted XREFs are so common that they are usually transparently recovered by all PDF readers, even picky ones such as PDF.JS. However, your pdflatex may generate a document based on the optimized PDF 1.5 specification, where the XREF is stored not in cleartext as in PDF 1.4, but rather as a separate, compressed object. This configuration choice is made for the user by the TeX distribution, so even a freshly updated pdflatex install may generate PDF 1.4 documents.

Even when compressed, corrupted XREFs are recovered by some readers, such as GS and Sumatra. Unfortunately, Foxit, Adobe, Firefox, Chrome, and Poppler-based readers—such as Evince and Okular—would reject such a document. Although rejecting corrupted documents out of hand is the best strategy, even Pastor Laphroaig would be pretty pissed if folks couldn’t read his epistles because of this.

A simple and elegant workaround that achieves 100% reader compatibility with our MBR PDF is to make sure that, even if your pdflatex distribution generates a 1.5 format document, it doesn’t compress the XREF. This is easily done by adding the following command to your \LaTeX source.

\pdfobjcompresslevel=0

This command will cause pdflatex to store non-objects uncompressed while still taking advantage of other 1.5 features such as reducing document bloat. I should add that, although the fix looks trivial, finding the real cause and the most elegant solution was a challenge.

---

Enjoy booting this PDF, and be sure to share copies—both electronic and paper—so that your neighbors can enjoy it as well!
CALC.EXE GTFO
9 A Vulnerability in Reduced Dakarand from PoC∥GTFO 01:02

by joernchen of Phenoelit

I’m not a math guy, so this is a poor man’s RNG analysis. Try it yourself at home!

9.1 Introduction

In PoC∥GTFO 01:02, Dan Kaminsky proposed the following code for use as a Random Number Generator, arguing that the phase difference between a fast clock and a slow clock is sufficient to produce random bits in a high level language. This is a reduced version of his Dakarand program, with the intent of the reduction being that if there is any vulnerability within the code, that vuln ought to be exploitable.

```javascript
// 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();
```

Actually the above code boils down to the function flip_coin, which takes a boolean value n=0 and continuously flips it until the next millisecond. The outcome of this repeated flipping shall be a random bit. We neglect the get_fair_bit function mostly in this analysis, as it just slows down the process and adds almost no additional entropy. For gathering random bits we are just left with the clock ticking for us.

9.2 A Naive Analysis

In order to analyze the output of the RNG we need some of its output, so I simply put up a small HTML piece which would pull out 100.000 random bytes out of the above RNG and log it to the HTML document. Then a severe 90-minute DoS on my Firefox 24 happened, after which I managed to copy and paste one hundred thousand uint8_t results into a text file.

After messing with several tools like ministat, sort and uniq I could show with the following ruby script that this RNG (on my machine) has a strong bias towards bytes with low hamming weights:

```ruby
#!/usr/bin/env ruby
f=File.open(ARGV[0])

h = Hash.new
f.each_line do |m|
    n = m.to_i
    if h[n].nil?
        h[n]=1
    else
        h[n] = h[n]+1
    end
end

t = h.sort_by do |k,v| v end
```

```
t.each do |a|
  puts "Num: \#{a[0]} + "
  puts "Count: \#{a[1]} + "
  puts "Weight: \#{a[0].to_s(2).split('').reject{|j|j=='0'}.count}" end

The shortened output of this script on the 100k 8bit numbers is as follows. Note that the heavy
hamming weights, like 11111111 are least common and the light hamming weights, like 00000000 are
most common.

<table>
<thead>
<tr>
<th>Value</th>
<th>Count</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>22</td>
<td>8</td>
</tr>
<tr>
<td>254</td>
<td>23</td>
<td>7</td>
</tr>
<tr>
<td>251</td>
<td>28</td>
<td>7</td>
</tr>
<tr>
<td>253</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>127</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>239</td>
<td>34</td>
<td>7</td>
</tr>
<tr>
<td>230</td>
<td>34</td>
<td>7</td>
</tr>
<tr>
<td>223</td>
<td>36</td>
<td>7</td>
</tr>
<tr>
<td>247</td>
<td>37</td>
<td>7</td>
</tr>
</tbody>
</table>

... ... ...

| 132   | 1173  | 2      |
| 64    | 1821  | 1      |
| 32    | 1881  | 1      |
| 16    | 1922  | 1      |
| 1     | 1934  | 1      |
| 8     | 2000  | 1      |
| 4     | 2042  | 1      |
| 2     | 2133  | 1      |
| 128   | 2145  | 1      |
| 0     | 3918  | 0      |

The table lists the Number which is the output of the RNG along with this number’s hamming weight
as well as the count of this number in total within the 100,000 random bytes. For a random distribution
of all possible bytes we could expect roughly a count of 390 for each byte. But as we see, the number 0
with the hamming weight 0 peaks out with a count of 3918, whereas 255 with the hamming weight of 8
is generated 22 times by the RNG. That’s not fair!

9.3 My fair bit is not fair!

Real statistical analysis of an RNG is hard, and I will not attempt it here.
Still, looking at a few simple distributions might give us a hint (alas, only a
hint) of what might behind the unfairness.

First, a short recap on how this RNG works:

We’ve got a 1 millisecond timeslot from t0 to t1, where at t1 the flip_coin
method will stop. The first call to get_random_byte can happen anywhere
between t0 and t1:

Somewhere here the JS engine jumps in

Let’s say it is here:

Now the algorithm happily flips the bit until t1 and hands over the result
of this flipping as a random bit (note that we’re omitting get_fair_bit here).
Although we cannot predict the output of a single run of flip_coin, things get a bit more predictable when we make a lot of consecutive calls to flip_coin. Let’s say we need the time d to process and store the result of flip_coin. So the next time we flip_coin we are at t1 + d1:

Now the RNG flips the coin until t2 in order to give us a random bit. As we are calling the RNG more than twice in a row, the next flip_coin is at t2 + d2, and so on.

The randomness and fairness of the RNG’s random bit depends on how fairly and randomly we get odd and even values of d, since that the same amount of flips yields the same bit as we have a static start value of 0/false. So it makes sense to look at the distribution of d. To visualize this and to compare it with another browser I came up with this slight modification of the RNG that counts the flips and records them right inside the HTML page:

```javascript
function flip_coin()
{i=0;n=0; then=millis()+1;
 while(millis()<=then) {n=!n; i++}
 return [n,i];}

function get_fair_bit()
{while(1) {a=flip_coin(); if(a[0]!=flip_coin()[0]) {return(a);}}}

function doit(){
 var i = 10000;
 while(i--){
 var d = document.getElementById('target');
 var content = document.createTextNode(get_fair_bit().toString() + '\n');
 d.appendChild(content);
 }
}
```

Loading the page in Chromium and Firefox and throwing them into gnuplot, we get:

![Graphs showing the distribution of flip_coin counts in Chromium and Firefox](image)

We can see that the graph for Chromium has a lot more variance in the number of coin flip within a millisecond than that for Firefox. Although, strictly speaking, it might still be possible to get good randomness with poor variance if the few frequent values were to alternate just so due to some underlying scheduling magic, it seems reasonable to expect that the same magic would also increase the variance in the flip numbers.

We can also see, with the help of simple UNIX tools, that Chromium counts do not peak out to a certain value, unlike those of Firefox:

\[11\] The second coin flip in get_fair_bit complicates it a bit, but it cannot substantially improve the RNG’s entropy if it lacks in the first place.
9.4 Closing words

In conclusion we see that in Firefox under stress Dan’s RNG appears to fail at exactly the point he wanted to use as the main source of randomness. The tiny clock differentials used to gather the entropy are not given often enough in Firefox. There is still much room to stress this RNG implementation. Bonus rounds would include figuring exactly what the significant difference between the Firefox and Chromium JavaScript runtime is that causes this malfunction on Firefox. Also attacks on other JavaScript runtimes would be interesting to see. It might even be the case that this implementation has different results under different conditions with respect to CPU load.

A broader question occurs: The Dakarand RNG relies on what could be called a “code clock.” It may be that in many kinds of environments stressed code clocks tend to go into phase with one another. Driven by stress to seek comfort in each other’s rhythms, their chance encounters may grow into something more close and intimate, grinding into periodic patterns. Which, of course, is bad for randomness. Can we learn to tell such environments from others, where periodization with stress doesn’t happen? –PML
This page intentionally left blank.
Draw your own damned picture.
10 Juggernauty

‘Twas UMBRA, and the STUNT WORMS
Did ZARF and CIMBRI in the SUEDE:
All GUPY were the PUZZLECUBES,
And the DIRESCALLOP AQUACADE.

“Beware the JUGGERNAUT, my son!
The RONIN bytes, the IMSI catch!
Beware the TUSKATTIRE, and shun
EGOTISTICAL GIRAFFE!”

He brought his FERRET CANNON forth:
yet SKOPE he not the RUTLEY spoor —
So browsed he to an onion,
And surfed awhile in Tor.

And, as in BOOTY Tor he surfed,
The JUGGERNAUT, with eyes of FLAME,
Leapt from the EVOLVED MUTANT BROTH,
with DISHFIRE as it came!

One, two! One, two! And through and through
The FERRET CANNON’s furred attack!
He left it dead, and with its LED
He rode his QUICK ANT back.

“And, has thou slain the JUGGERNAUT?
Come to my arms, my DANGERMOUSE!
OLYMPIC day! MESSIAH! MORAY!”
He TALKQUICK in his joy.

‘Twas UMBRA, and the STUNT WORMS
Did ZARF and CIMBRI in the SUEDE;
All GUPY were the PUZZLECUBES,
And the DIRESCALLOP AQUACADE.
“He that is without sin among you, let him first cast a stone at her.”
11 A Call for PoC

by Rt. Revd. 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!

11.1 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 implement Dakarand in a 512-byte boot sector; teach me how to compose shellcode in Korean characters; or, teach me how to patch Natalie’s Tamagotchi shellcode with nothing but MSPAINT.EXE. 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, I expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for our poor bastard of an editor to apply to 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.

11.2 Other Departments

<table>
<thead>
<tr>
<th>Editor at Large</th>
<th>Dept. of Bringing APT Home</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept. of Funky File Formats</td>
<td>Dept. of Fail</td>
</tr>
<tr>
<td>Ethic Board</td>
<td>Dept. of Busting BS</td>
</tr>
<tr>
<td>Poet Laureate</td>
<td>Dept. of Drama</td>
</tr>
<tr>
<td>Dept. of PHY</td>
<td>Dept. of PHY</td>
</tr>
</tbody>
</table>

Rt. Revd. Pastor M.L.  
Cultural attaché of the 41st Directorate  
Ange Albertini  
FX of Phenoelit  
The Gruq  
pipacs  
Ben Nagy  
Xbf  
Michael Ossmann

32