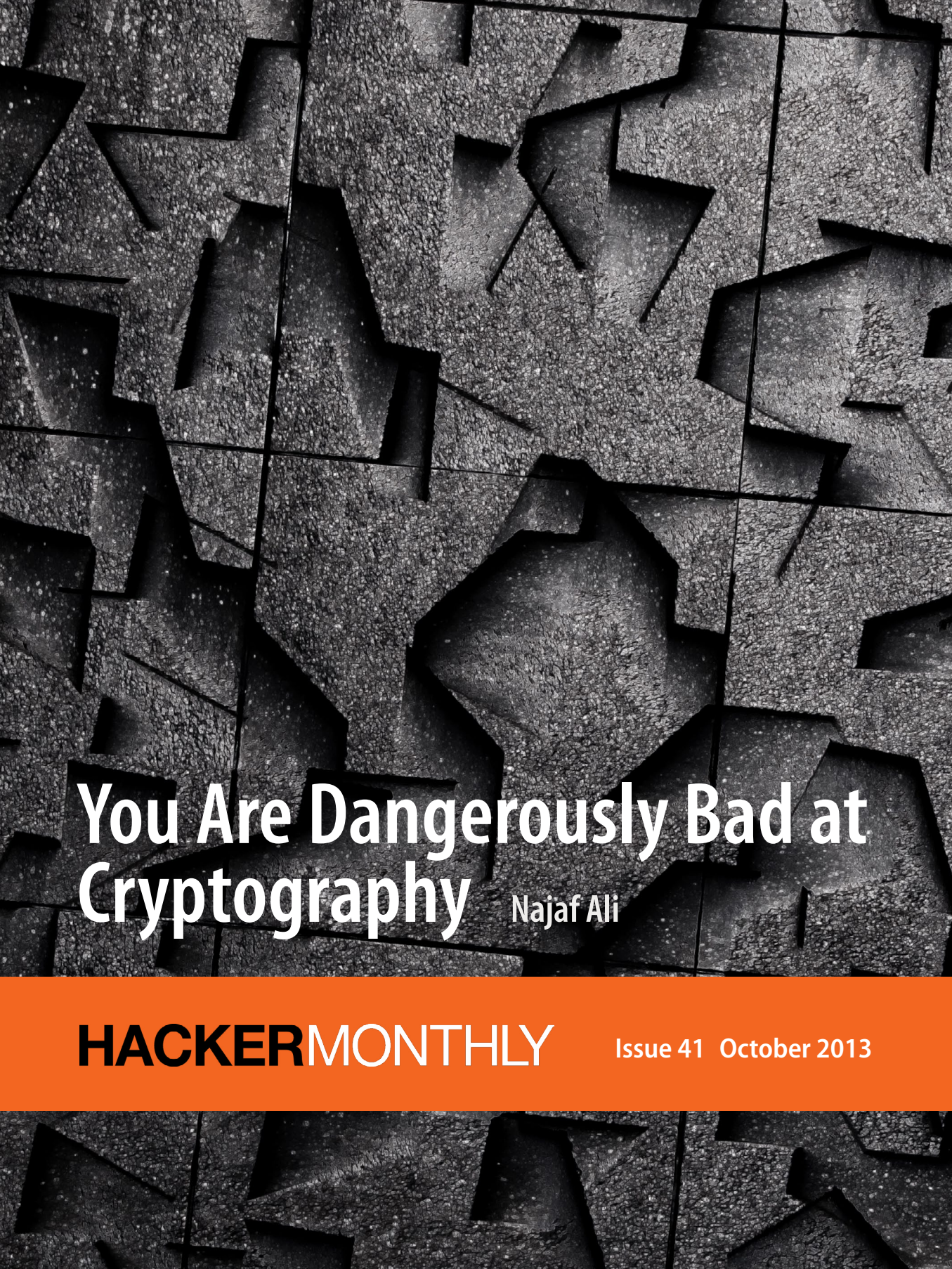


You Are Dangerously Bad at Cryptography

Najaf Ali

HACKERMONTHLY

Issue 41 October 2013

mailjet

EMAIL FOR YOUR APPS

SEND. TRACK. DELIVER.

Your one stop shop for **ALL your email needs.**

Manage lists as well. No extra fees for **Newsletters.**

Priority headers to deliver notifications in **real time.**

Go for

mailjet.com

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

Curator
Lim Cheng Soon

Contributors
Ram Rachum
Najaf Ali
Sacha Chua
Kyle MacDonald
Jeff Escalante
Craig Gidney
Jason Cohen
David Cain
Shaanan Cohney

Proofreaders
Emily Griffin
Sigmarie Soto

Illustrator
Stefan Hartmann

Ebook Conversion
Ashish Kumar Jha

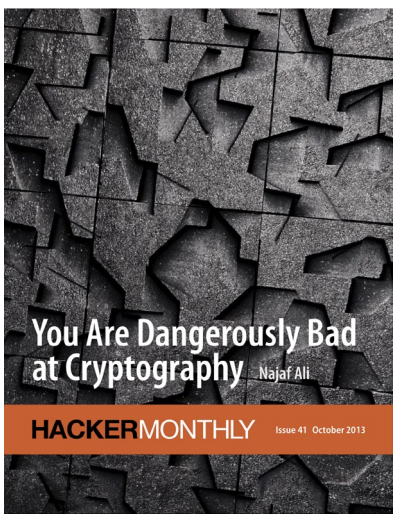
Printer
MagCloud

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: Stefan Hartmann

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-41*

Indescribable Numbers

The theorem that made me fall in love with math

By RAM RACHUM

Let me open this blog post by quoting a Zen koan from Mumonkan, along with the comments and poem that Zen master Mumon added to it:

Koan:

A monk asked Zen master Nansen, "Is there any teaching no master has ever taught before?"
Nansen replied, "Yes, there is."
"What is it?" asked the monk.Nansen answered, "It is not mind, it is not Buddha, it is not things."

Mumon's comments about the koan:

Being asked a question, old Nansen gave away his treasure words. He must have been greatly upset.

Mumon's poem about the koan:

Nansen was too kind and lost his treasure,
Clearly words have no power.Even though the mountain becomes the sea,
Words cannot open another's mind.

Now that I've put you in the mood, I'd like to tell you the story of my theorem about "indescribable numbers." I can't believe I didn't write about this thing years ago. This theorem was one of the most exciting episodes in my love affair with math, and it's a tale worth telling.

The year was 2005. I was about 19 years old, with only a general and vague idea of what I wanted to do with my life. I enrolled to study Electrical Engineering at the Technion, a leading university in Israel. Why did I choose Electrical Engineering? Because (a) it

sounded science-y and (b) it was said that there was a lot of money in it.

Contrasting my present self with my 2005 one, the most striking thing is how empty my mind was from the various strong opinions and views that inhabit it today. Not quite the emptiness of mind that a Zen master would aspire to, but rather an emptiness in which the two promises above, of scienciness and money, were all I needed to decide that EE was indeed for me, at least for a little while.

One morning I was walking to class, or perhaps I should say climbing; The Technion, built upon Mount Carmel, is famous for its brutally steep slopes. I lived in the Canada dorms, which were located at the lowermost point in the campus, and I had to walk up all the way to the campus' vertical center. I'm not the athletic type, so this daily ordeal left me sweaty and tired every time I got to class.

Just as I was passing the faculty for Civil Engineering on my right, the first beads of sweat already formed on my forehead, a funny thought came to me.

That thought was: **Could there be numbers that cannot be described?**

I should explain this idea in detail.

Imagine that you're solving a homework problem in physics. Say the problem introduces you to a helium-filled balloon as it is making its way towards the ceiling, and you are tasked with finding its Y coordinate at t equals 3 seconds, air resistance neglected. After some work you come up with an answer. If the homework in question is given in junior high or high school, the resulting number describing the height is likely to be a simple one. Perhaps 7 meters, or 11 meters; thanks are due to the invisible compassion of high school physics teachers who have a hard enough time as it is teaching physics to teenagers who are much more interested in sex. The teachers, knowing they have only a precious modicum of the students' attention at their disposal, compose their problems in such a way that the final number would be so simple, usually an integer and in wild cases a simple fraction, that when the student finally hits upon it, he can put his mind at ease. After all, if after these complex and error-prone calculations, fraught with quadratic equations, you got a result of 7 meters, that's a good enough reassurance as any that an error was not made and that the answer you came upon is the correct one.

As high school pupils become university students and are gradually weaned from the soft and cozy world of integers, the numerical solutions gradually become more complex.

You could be hard at work on a more advanced variant of the above physics problem in Physics 101, air resistance emphatically not neglected, only to arrive at a result of 3*Sin(57) meters, which the last few pages of the book reassure you is the correct answer.

You stare at that number, 3*Sin(57), or 2.516011703836(...) in all its decimal glory, and you imagine the fictional God of that fictional universe who placed each and every one of the sextillions of helium atoms in that balloon in just the right position, so that when t equals 3 seconds, the height of the center mass of the balloon would be at precisely 3*Sin(57) meters, to the last of its infinite digits. And you realize, of course, that this invisibly compassionate God is nothing more than a braver version of the invisibly compassionate high school teacher, who was so careful not to frighten you with an unholy number such as -6.33333.

So you imagine what a real God would do, and what the real height of a real helium balloon would be. Since we're dealing with pure math, where two numbers would be absolutely different even if they differ from one another only in the billionth digit, it is obvious that physical experiments are not going to help us here. We turn to thought experiments instead.

We imagine a real God, creating a real balloon with sextillions of real helium atoms, each of which is emphatically apathetic to our desire for beautiful numbers as our solution. And we imagine that balloon at t equals 3 seconds. We think, what would be its height? Definitely not 3*Sin(57) meters. There is no chance in hell that the solution would be as pretty as that. In fact, intuition tells us that the number would be as ugly a number as we've ever seen. Next to that number, 3*Sin(57) or even 3*Sin(57)^74*Arctan(0.42^3.5) would look as pathetically simple as 7. No, that number would be so ugly, that…that there won't even be a way to describe it. And now we're finally reaching the original thought with which I opened this discussion: indescribable numbers.

Let me humbly take upon myself the job of being that real God of that apathetic universe; and let me hereby declare, with the full power of my omnipotence, that the height of the balloon in meters is, as suspected, not 3*Sin(57), but rather the following number:

3.7493685096834712522039(...)

Yes, be impressed by the glory of these digits that I have created in my infinite wisdom; look at them and wonder what pattern might lie behind them, knowing full well that I am too terrible a God to have planted there any pattern you might be able to understand with your limited human mind; stare imploringly into the siren call of the final ellipsis, for you know that no matter how often you expand it, I will always smile when giving you more and more digits, because you and I will both know that the final wisdom of The Number will always be mine and never yours.

Please excuse me, it's not that often that I get to be an omnipotent being and I'd like to have fun with it as long as it lasts. I'll retire now and join you in the pathetic human race.

So the question is: Are there really such numbers as 3.749(...), whose pattern is not only unknown to us, but could never be known and will forever be beyond the grasp of us mere mortals? Could there be a number that simply cannot be described by any means known to us? A number which is definitely not an integer, obviously not rational, desperately not algebraic; we know that it is at some innocuous-looking spot on the line of real numbers, somewhere between honest old 3.749 and honest old 3.750, but we know that it is a different creature entirely than these two model citizens of the Reals. Our number carries secrets infinitely more profound, and therefore we know that unlike the above duo, we'll never be able to pinpoint the exact spot in which this number resides. Like a telephone number that is unlisted in the Yellow Pages; if God didn't give you the number as he spoke from behind a burning bush, you'll never find it using the scientific method.

Those were my thoughts as I was wiping sweat off of my forehead and passing the faculty for Civil Engineering. Being 19, my mathematical mind not yet fully developed, I thought: What an interesting philosophical conundrum. That's pretty cool. I doubt it could ever be solved though; yet another unanswered, vague philosophical question in the long history of unanswered, vague philosophical questions. I got to class, I believe it was Calculus, and concentrated on that instead.

Fast forward a few months.

I was into my second semester by now, which would prove to be my final semester. I had a little bit more training in math, given to me not only in the courses I've taken, but also from random math-related articles I would read on the web.

It was morning again.

I was taking my usual morning climb from Canada dorms to class, passing Civil Engineering on my right.

That is when the answer came to me. As Zen master Mumon would say: **At that moment, I became enlightened.**

I found an answer to the question of indescribable numbers, and I found a **mathematical proof** to the answer. I managed to take this philosophical question, so vague and soft, and not only define it mathematically, but find *an actual goddamned water-tight proof* to validate my answer.

I was amazed. I was shaken. I could hardly believe this was really happening.

The answer to the question is this: Yes, there are such things as indescribable numbers. In fact, **the vast majority of real numbers are indescribable, and only a tiny fraction of real numbers are of the familiar describable variety.**

Holy shit.

I didn't really know what to do with this. Is my proof even right? Did I just make a first-rate scientific discovery? Perhaps I've rediscovered a known truth, perhaps a theorem my Calculus 3 teacher will teach to us next semester as my classmates will yawn and doodle in their notebooks? Perhaps I am simply deranged?

Then I got an idea who to turn to: My Calculus 2 teacher. He was a friendly old man. He was clumsy, had a huge unkempt beard, and his great love and fascination for math showed through in our Calculus 2 lectures. He was your classic math professor. I would later learn that he was originally from Australia, which helped explain how clumsily energetic he was.

I found his love for math contagious, even when it was hard for me to keep up with the technical aspects, which was often. My fellow Electrical Engineering students, more enchanted by the promise of great money in EE rather than a love for science, were less cooperative with him. It was sad to see someone so passionate trying to inspire those who simply did not want to be inspired. (No disrespect intended to these students; I love money too.)

After the class ended, and all the students were putting their notebooks back in their backpacks and leaving, I went to the professor's desk. He was packing as well. I told him I had a mathematical thought that I didn't really know what to do with. He was intrigued. He was too busy at that time to continue the conversation, but he told me to email him, and via email we set a time for an appointment.

I arrived at his office. It was small and cramped and incredibly messy. There were boxes of papers everywhere. Fortunately, there was a whiteboard.

I was excited. I've never been to a professor's office before. I've never had a serious conversation about math before with someone who could be actually considered an expert in it. I tunneled my energies of excitement to making sure that I'm explaining the original question and my proof in a clear, calm, but relentlessly watertight way.

I will recreate the explanation of the proof for you now, leaving out the more technical parts.

The proof is not complex; any BSc of Mathematics would understand it quite easily.

The only way us humans have of describing numbers is using language. Sometimes that could be a non-math-specific language like English. For example, "seven" is a description of 7; "the sum of ten and three" is 13; and "the ratio of a circle's circumference to its diameter" is a concise expression of Pi.

Quite often though, we prefer to describe numbers using mathematical language. "7" is a simple number described by that language. Our previous acquaintance "3*Sin(57)" is a more complicated one, and "3*Sin(57)^74*Arctan(0.42^3.5)" is an even more impressive specimen.

This mathematical language is more powerful for describing numbers than the English language, but it's still a language; every such description of a number is just a finite string of symbols, taken from a finite pool of available symbols: "3", then "*", then "S", then "i", then "n", then "(", then "5", then "7", and finally ")". Descriptions can get much longer than that, as long as they're finite. But there is an infinite number of descriptions, since we can combine symbols in infinite many ways to make descriptions as long as we want.

Now this is where things start to get interesting. I'll try not to get too technical.

In math, we have several different kinds of infinity. The smallest kind is "countable infinity" also known as Aleph zero. It is the infinity of natural numbers, the infinity of 0, 1, 2, 3…

The infinity just a step bigger than it is Aleph one, the infinity of real numbers: The infinity of an impossibly dense line of numbers, between each two, no matter how close, resides yet another infinite spectrum of numbers, itself bigger than the previously mentioned infinity of natural numbers.

It is well proven that Aleph one, which is the infinity of the real numbers, is undeniably bigger than the infinity of the natural numbers. What this means, is that **you can never "cover" the real numbers with the natural numbers.** In technical terms, you can't have a surjective mapping from the set of naturals to the set of reals. In more intuitive terms, if you try to pair up each natural number to a real number, you will run out of natural numbers way before you'll run out of real numbers. (We can never really imagine the moment where we've run out of natural numbers, since there are an infinity of them… But bear in mind that the set of real numbers is "even more infinite," and that's the closest I can give you to an intuitive description.)

Let's get back to describing numbers. We've said that our descriptions of numbers using mathematical language are nothing but finite strings of a finite language. There's infinitely many of them, but that infinity can be easily shown to be Aleph zero, the smallest infinity. Just consider that any such description, like "3*Sin(57)", could be saved as a text file on a computer, and every file on a computer is just ones and zeroes.

However, the real numbers have the bigger infinity of Aleph one. You can feel the proof forming, can't you?

When we take a description like "3*Sin(57)", we know it has an obvious counterpart in the set of real numbers: The number it's describing, the very real number close to 2.51. We can in fact pair every such description to the number it describes. "7" could be paired with the number 7, and the venerable "3*Sin(57)^74*Arctan(0.42^3.5)" will also be paired with the number it describes. We're in effect pairing up each member of the set of descriptions to a member in the set of real numbers.

But, remember what we said before, that if you try to pair up an infinite set of size Aleph zero with an infinite set of size Aleph one, you will never be able to cover the entire Aleph one set. There will always be unfortunate members in that bigger set that would be left without a counterpart in the smaller set.

Those unfortunate members are indescribable numbers.

Why? Because consider what they are: They are real numbers, for which we have just proven it is impossible to find a description that will match them. We have proven that no description will ever describe them.

We have proven that indescribable numbers exist. Q.E.D., and please someone hand me a cigarette.

I explained all this to the professor. He asked a bunch of questions, and I managed to answer all of them without having an "Oh shit, you just discovered a critical flaw in my proof, my entire proof is wrong now fuck me" moment. At the end, when he was quite convinced that my proof was correct, he said something along the lines of "that's pretty cool." He said he was unfamiliar with this area of mathematics, but that he thought my proof was correct and really interesting.

I felt so proud. I managed to find a proof that impressed not only me, but a genuine crazy math professor! I'm smart!!!

The professor arranged an appointment for me with a different professor who did specialize in logic, and was familiar with my theorem. He said that my theorem and proof have been well known to logicians since the 1940s. He was still impressed that I was able to prove them despite being a first-year Electrical Engineering student.

I was a bit saddened that my theorem was old stuff for logicians and not a new discovery. There goes my Nobel Prize…

But I was still so happy just to have found the proof. The experience of having taken a vague philosophical question, and using the precise machinery of math to state it rigorously and then actually prove it, was amazing for me. It was like discovering a net with which I managed to catch a beautiful butterfly. Imagining those indescribable numbers out there, the mathematical equivalent of dark matter, occupying most of the space in the set of real numbers despite being completely invisible and unattainable… I was in love with math.

That episode was one of the reasons that I quit Electrical Engineering and spent the next 2 years of my life studying mathematics. But that's another story :) ■

Ram Rachum is a freelance software developer based in Tel-Aviv, Israel. He works almost exclusively in Python. He loves nothing more than stripping bullshit away until all that is left is the unembellished truth.

Reprinted with permission of the original author.
First appeared in *hn.my/indescribable* (ram.rachum.com)

You Are Dangerously Bad At Cryptography

By NAJAF ALI

THE FOUR STAGES of competence:

1. **Unconscious incompetence**: When you don't know how bad you are or what you don't know.
2. **Conscious incompetence**: When you know how bad you are and know what steps you need to take to get better.
3. **Conscious competence**: When you're good and you know it (this is fun!)
4. **Unconscious competence**: When you're so good you don't know it anymore.

We all start at stage one whether we like it or not. The key to progressing from stage one to stage two in any subject is to make lots of mistakes and get feedback. If you're getting feedback, you begin to create a picture of what you got right, what you got wrong and what you need to do better next time.

Cryptography is perilous because you get no feedback when you mess up. For the average developer, one block of random base 64 encoded bytes is as good as any other.

You can get good at programming by accident. If your code doesn't compile, doesn't do what you intended it to or has easily observable bugs, you get immediate feedback, you fix it and you make it better next time.

You cannot get good at cryptography by accident. Unless you put time and effort into reading about and implementing exploits, your home-grown cryptography-based security mechanisms don't stand much of a chance against real-world attacks.

Unless you pay a security expert who knows how to break cryptography-based security mechanisms, you have no way of knowing that your code is insecure. Attackers who bypass your security mechanism aren't going to help you with this either (their best case is bypassing it without you ever finding out).

Take a look at some examples of misused crypto below. Ask yourself, if you hadn't read this post, would you have caught these errors in real life?

Authenticating the API for your photo sharing website

Message Authentication with md5 + secret

Once upon a time, a photo sharing site authenticated its API with the following scheme:

- Users have the following two credentials:
  - A public user id that they use to identify themselves (safe to send in the clear)
  - A shared secret that they use to sign messages (must be kept private)
- The user makes API requests over HTTP/HTTPS (it doesn't matter). Destructive changes are made using a POST/GET request with specific parameters (e.g. { action: create, name: 'my-new-photo' }).
- To authenticate the message, the user sends their user id as a parameter, and then signs the message with their secret key. The signature is the md5 of the shared secret concatenated with the key-value pairs.

To check that the client is the user he claims to be, the server generates the signature from the request parameters and the secret key it has on file for that user.

The code for this could be:

```
# CLIENT SIDE
require 'openssl'

## Our user credentials
user_id = '42'
secret  = 'OKniSLvKZFkOhlo16RoTDg0D2v1QSBQvG1hHflMeO77nWesPW+YiwUBy5a'

## The request params we want to send
params = { foo: 'bar', bar: 'baz', user_id: user_id }

## Build the MAC
message      = params.each.map { |key, value| "#{key}:#{value}"
}.join('&')
params[:mac] = OpenSSL::Digest::MD5.hexdigest(secret + message)

## Then send the request via something like...
HTTP.post 'api.example.com/v3', params

# SERVER SIDE
## Grab the user credentials out of the DB
user   = User.find(params[:user_id])
secret = user.secret

## Get the MAC out of the request params
challenge_mac = params.delete(:mac)
## Calculate the MAC using the same method the client uses
message        = params.each.map { |key, value| "#{key}:#{value}"
}.join('&')
calculated_mac = OpenSSL::Digest::MD5.hexdigest(secret + message)

## Compare the challenge and calculated MAC
if challenge_mac == calculated_mac
  # The user authenticates successfully, do what they ask
else
  # The user is not authenticated, fail
end
```

With a basic understanding of how md5 works, this is a perfectly reasonable implementation of API authentication. That looks secure, right? **Are you sure?**

It turns out that this scheme is vulnerable to what's called a length extension attack.

Briefly:

- If you know the value of `md5('foo')`, due to the way md5 works, it's trivial to compute `md5('foobar')`, without knowing the prefix "foo".
- So if you know the value of `md5('secretfoo:bar')`, it's trivial to compute `md5(secretfoo:bar&bar:baz)` without knowing the prefix "secret".
- This means that as long as you have one example of a signed message, you can forge signatures for that message plus any arbitrary request parameters you like and they will authenticate under the above described scheme.

Any developer who didn't know about this beforehand would have easily been caught out. The developers at Flickr, Vimeo and Remember the Milk rolled this out to production.

The point isn't that you should know about every esoteric detail of the internals of cryptographic functions. The point is there are a **million ways to mess up cryptography**, so don't touch it.

Not convinced? OK, let's try fixing this example and see if we can make it secure...

Message Authenticating with HMAC

You hear about this security vulnerability via your friendly neighbourhood whitehat and he recommends that you use a Hash-based Message Authentication Code or HMAC to authenticate your API requests.

Great! HMAC's are designed for our use case. This is a drop-in replacement for what you were doing to verify the signature before. Our server verification code can now look like this:

```
require 'openssl'

## Grab the user credentials out of the DB
user   = User.find(params[:user_id])
secret = user.secret

## Get the MAC out of the request params
challenge_mac = params.delete(:hmac)

## Calculate the HMAC
## We'll do the same thing on the client when we
## generate the challenge
message         = params.each.map { |key, value| "#{key}:#{value}"
}.join('&')
calculated_hmac = OpenSSL::HMAC.hexdigest(OpenSSL::Digest.new('md5'),
secret, message)

## Compare the challenge and calculated MAC
if challenge_hmac == calculated_hmac
  # The user authenticates successfully, do what they ask
else
  # The user is not authenticated, fail
end
```

That looks secure, right? **Are you sure?**

It turns out that the verification code above is vulnerable to a timing attack that allows you to guess the correct MAC for a given message.

Briefly:

- For a given message, attempt to send it with a HMAC of all one single character. Do this once for each ASCII char (e.g. "aaaa...", "bbbb...", etc.).
- Measure the time each request takes to complete. Since string equality takes a tiny bit longer to complete when the first char matches, the message that takes the longest to return will have the correct first character.
- Smooth out noise from latency in two ways:
  - Run a couple of hundred or thousand requests for each guess to get an average time.
  - Run your timing attack code from within the same data centre. If you're having trouble determining the data centre, in the worst case you can spin up a box at each of the major providers and find out which box takes significantly less time to ping the target server.
- Once you've determined the first character, repeat for the second by changing the second char onwards (e.g. if "x" is the first char, try "xaaa...", "xbbb...", etc.).
- Keep going until you have the whole HMAC.
- Using the above defined technique, you can reliably determine the HMAC of any message you want to send to the API and authenticate successfully.

Again, perhaps you didn't know about timing attacks and you're not expected to. The point isn't that you should have known the details of specific vulnerabilities and watched out for them. The point is that there are a million ways to mess up cryptography, so don't touch it.

All the same, let's go ahead and try to make this more secure...

Verifying HMACs in a time-insensitive way

You get around timing attacks by comparing the sent and computed MAC in a time-insensitive way. This means you can't rely on your programming languages built in string equality operator, as it will return immediately when it finds a single character difference.

To compare strings, we can take advantage of the fact that any byte XORed with itself is 0. All we have to do is XOR each byte from string A with the corresponding byte from string B, sum the resulting bytes and return true if the result is 0, false otherwise. In Ruby, that might look like this:

```
require 'openssl'

## Time insensitve string equality function
def secure_equals?(a, b)
  return false if a.length != b.length
  a.bytes.zip(b.bytes).inject(0) { |sum, (a, b)| sum |= a ^ b } == 0
end

## Grab the user credentials out of the DB
user   = User.find(params[:user_id])
secret = user.secret

## Get the MAC out of the request params
challenge_hmac = params.delete(:hmac)

## Calculate the HMAC
## We'll do the same thing on the client when we generate
## the challenge
message         = params.each.map { |key, value| "#{key}:#{value}"
}.join('&')
calculated_hmac = OpenSSL::HMAC.hexdigest(OpenSSL::Digest.new('md5'),
secret, message)

## Compare the challenge and calculated MAC
if secure_equals?(challenge_hmac, calculated_hmac)
  # The user authenticates successfully, do what
  # they ask
else
  # The user is not authenticated, fail
end
```

That looks secure, right? **Are you sure?**

I doubt it. It marks the edge of my knowledge in terms of potential attack vectors on this sort of scheme, but I'm not convinced that there's no way to break it.

Save yourself the trouble. Don't use cryptography. It is plutonium. There are millions of ways to mess it up and precious few ways of getting it right.

P.S. If you must verify HMACs by hand and you have activesupport handy, you'll get that time-insensitive comparison from using `ActiveSupport::MessageVerifier`. Don't code it from scratch, and for crying out loud don't copy-paste my implementation above.

P.P.S. Still not convinced? Do the Matasano Crypto Challenges [hn.my/matasano] and see if that doesn't change your mind. I'm not half way through and I've already had to get in touch with two former clients to fix their broken crypto. ■

Najaf Ali is an independent technical consultant based in London, UK. He helps startups and small businesses build better software.

Reprinted with permission of the original author.
First appeared in *hn.my/crypto* (happybearsoftware.com)

Illustration by Stefan Hartmann.

Breaking a Toy Hash Function

By CRAIG GIDNEY

You probably know that hash functions can be used to protect passwords. The idea is that someone with access to the hash can't figure out the corresponding password, but can use the hash to recognize that password when it is received. This is really, really useful in cases where attackers have access to your source code and your data.

For example, consider WarCraft 3 maps (essentially little self-contained games). Maps specify terrain, units, code, etc. but can't access the internet, the file system, or even the current time. Anyone who has a map knows every detail of how it works, if they care to look. If you want to make a map that recognizes a password, perhaps to give yourself some sort of unfair admin powers as a joke, you'll want to protect that password so that people who look inside the map won't be able to play the joke on you.

In fact, years ago, I happened across exactly that sort of thing: a bit of JASS code that hashes the user's name and a password in order to recognize the map maker and a couple of their friends. However, the hash function being used was created by a friend of the map maker. It is not a standard cryptographic hash function.

One of the standard refrains in cryptography is "Do not write your own crypto." Given that this person wrote their own crypto, I wondered if I could break their hash function. I tried a bit and gave up, but the problem stayed in the back of my mind. Every year or so I'd get the urge to go back and try again, waste a day messing with it, and give up again.

This year, I finally succeeded. I reversed the password, and all three usernames.

Note that I am not a cryptographer. The way I broke this function is probably… naive. I assume that, to a real cryptographer, this function is a toy to be crushed in an hour ("Ha! Just use X!").

Nevertheless, I broke the hash function and I'm going to explain how.

The Hash Function

Given that most readers won't know the intricacies of obfuscated JASS, I've taken the liberty of translating the hash function to C#:

```
static Tuple<Int32, Int32> Hash(string text) {
    var charSet="abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
`~!@#$%^&*()_+-=|[];',.{}:<>? ";
    Int32 a = 0;
    Int32 b = 0;
    foreach (var letter in text) {
        var e = charSet.IndexOf(letter);
        if (e == -1) e = charSet.Length + 1;
        for (var i = 0; i < 17; i++) {
            a = a *-6 + b + 0x74FA - e;
            b = b / 3 + a + 0x81BE - e;
        }
    }
    return Tuple.Create(a, b);
}
```

As you can see, the state of the hash function is made up of two 32-bit signed integers (a,b) that both start out as 0. The input is a sequence of characters, drawn from 93 possibilities. Each character from the input is mixed into the state over a progression of 17 rounds and, when the last character has been mixed in, the result is just the final state of (a,b).

Note that addition and multiplication are unchecked (e.g. Int32.MaxValue+1 = Int32.MinValue, Int32.MaxValue*2 = -2) and division rounds towards 0 (e.g. -4/3= -1, 7/3 = 2).

In addition to the hash function, here is translated code to verify that a username/password combination is valid:

```
static bool Verify(string username, string password)
{
    var expectedPassHash = Tuple.Create(-0x20741256,
-0x4A579222);
    var expectedNameHashes = new[] {
        Tuple.Create(-0x52BEB283, -0x733C9599),
        Tuple.Create(0x605D4A4F, 0x7EDDB1E5),
        Tuple.Create(0x3D10F092, 0x60084719)
    };

    var passHash = Hash(password);
    var nameHash = Hash(username);
    return password.StartsWith("<+")
        && passHash.Equals(expectedPassHash)
        && expectedNameHashes.Contains(nameHash);
}
```

As you can see, both the valid usernames and the valid password are protected by hashing them. Also, the first two characters of the password are included in the code.

Side Note: Although it might seem dumb to give away some of the password's characters, it's actually a good idea given the context. The prefix is used as a filter for the chat event that triggers the hashing, to avoid hashing every single chat message said by anyone. The filter also allows the game to avoid secretly sharing all team messages with opponents. (They need to know something matching the filter was said in order to run the chat event trigger, and they need to know what was said in order to feed the right information into the hash function. Otherwise they can't advance in lockstep.)

Our goal is to find a username and a password that make Verify return true.

Leaking Entropy

The first thing to notice about the above function that suggests it should be easy to break, is that it leaks entropy. It is using non-reversible operations, which decrease the number of states the system might be in.

To make it easier to talk about that, here's a spread out version of the internal loop, with the multiplication by -6 factored and the division by 3 split into rounding followed by inverse-multiplying.

```
a *= 2;
a *= -3;
a += b;
a += 0x74FA;
a -= e;
b -= b % 3; // round to multiple
of 3, towards 0
b *= -1431655765; // multiplica-
tive inverse of 3 (mod 2^32)
b += a;
b += 0x81BE;
b -= e;
```

When working in modular arithmetic, some multiplications are reversible (do not leak entropy), but others aren't.

Multiplying a 32-bit integer by 3 does not decrease the amount of entropy because it is reversible. Every input state corresponds to exactly one output state. You can even efficiently run the operation backwards by multiplying by the modular multiplicative inverse of 3. The multiplicative inverse of 3 is $3 \cdot 1 = -1431655765 \pmod{2^{32}}$ because multiplying them together gives a result equivalent to one: $3 \cdot 3^{-1} = 3 \cdot -1431655765 = -42846795 = -2^{32} + 1 = 1 \pmod{2^{32}}$.

Multiplying by 2 is NOT reversible. It does decrease the amount of entropy. This happens because $(x + 231) \cdot 2 = x \cdot 2 + 2^{32} = x \cdot 2 \pmod{2^{32}}$, meaning both inputs of either x or $x + 2^{31}$ are collided into the single output of 2 * x. In the worst case this limits the possible number of output states to be half the number of input states, destroying 1 bit of entropy. Many inputs map to one output, so the operation is not reversible and leaks entropy.

The other non-reversible operation is rounding to the nearest multiple of 3 towards 0. In the worst case this destroys about 1.5 bits of entropy, reducing the number of possible states by about a third.

These leaks occur every single round, and it's possible for their cumulative effects to be very bad. It's a bit like those “mixing tank” problems you solve when learning differential equations, except the input mixture keeps changing color. If the tank is leaking then the contributions of the early colors to the average color decrease exponentially, instead of linearly, as more colors are added.

These leaks make me suspect that earlier values are in danger of "diluting away." Every round destroys a couple bits and replaces them with mixtures of the remaining entropy. Later values don't get destroyed and mixed much, but early ones do. Maybe, to find a preimage, I only have to care about the last few characters instead of all the characters. Maybe, to find a collision, I can significantly increase my chances by adding the same long suffix to any two starting strings.

It turns out that these leaks weren't devastating, but they really shouldn't have existed in the first place. Fixing the leak caused by multiplying by -6 is as easy as changing 6 to 7. Fixing the leak caused by rounding to a multiple of 3 is also easy: just remove the rounding.

Wait, no, that last idea is terrible.

Almost Linear

All of the operations in the hash function, except rounding to a multiple of three, are linear. They distribute over addition.

If we removed the rounding operation, the contributions of every input could be separated and reduced to a single multiplicative constant that depended only on the position relative to the end of the string. Each input value would be multiplied by the constant corresponding to its position, you'd sum up the products, and that'd be the result of hashing. Suddenly, finding an input that hashes to a given value would be like solving the subset sum problem, and there'd be all this structure we might be able to take advantage of to save huge amounts of time.

Fun fact: if you fixed the entropy leak due to the rounding (by removing it), but didn't fix the leak due to the multiplication by -6, you'd have made things far, far worse. The constants corresponding to positions would keep gaining factors of two. Ultimately, only the last four characters would get non-zero corresponding constants and collisions would be somewhat easier to find.

It's interesting that the operation that rounds b to be a multiple of three affects the state very little. It offsets it by at most 2, but that little tweak is the only reason reversing the hash function is difficult. Of course, in a properly designed hash function, the non-linearities are reversible and their effects are not tiny tweaks to state (e.g. they might XOR a into b instead of adding a into b, presumably flipping half of b's bits).

The fact that the non-linearity is so small made me wonder if I could just apply integer programming to the problem. Presumably integer constraint solvers are super-fast when there's this sort of regularity. That did not go well.

Integer constraint solvers are not designed with modular arithmetic in mind. Every solver I used failed to

reverse even three of the seventeen rounds needed to process a single character, because the solutions required values that exceeded the solvers' valid range. Confusingly, the solvers mostly just claimed "no solution". The only solver that actually told me I was going out of range, instead of pretending there was no solution, was IBM's CPLEX. I hereby award them one competence point.

I also tried extracting the non-linearities by rearranging the code by hand. I took this way, way too far before giving up.

Meet in the Spring

Eventually, I figured maybe I should try the obvious thing and brute-force the answer.

First, I tried just enumerating all inputs. This starts getting pretty slow once you get to five characters, since there are 93 possibilities for each character and $93^5 = 6956883693 \approx 10^{10}$. With that many possibilities to check, every additional operation needed to check a single possibility is adding at least a second to your running time (and hashing involves hundreds of operations). At six characters that goes up to a hundred seconds per operation, and you'll be left waiting for days.

Second, I tried to meet in the middle.

Because the entirety of the hash function's state is used as its output, it's possible to run it backwards (this is slower, though). Just do the inverse of each operation. This allows you to explore both forwards and backwards, while trying to find common middle states.

To say that this gives a performance boost is a bit of an understatement. Instead of using almost a trillion hash operations to try all possible six character strings, we're only going to spend a million hash operations and a million reverse-hash operations. The million hash operations are used to try all possible three character prefixes, building a dictionary that takes a reached state and tells you the prefix that reaches it. The million reverse hash operations are used to try all possible three character suffixes, telling you which intermediate states can be reached by exploring backwards from the end state. If there's a path from the start point to the end point, then one of the states reached by traveling backwards will be in the dictionary, and you're done.

I used meeting in the middle to go from searching all five character strings to all six character strings. I didn't bother with seven because my machine would go out of memory trying to store all the four-character states.

Third, I decided to use a bloom filter instead of a dictionary to store the middle states. Now, instead of immediately getting a solution when I found a match in the middle, each match was a possible solution that I could verify later on by re-exploring the possible prefixes.

Why is it worth sacrificing the immediate result to go from three "cached" rounds to four cached rounds? Because every cached round is effectively a 100-fold speedup. I could even have gone to five cached rounds, if my machine had more than 4 gigs of memory (the bloom filters had to be quite large to accommodate the hundreds of millions of items while maintaining low false-positive rates).

Fourth, I tried tracking integer constraints. I knew a lot of constraints that intermediate states had to satisfy, so I checked them constantly and discarded states that didn't fit. When I measured how much this was reducing the search space, it was a staggering 50% per reverse-round. I assumed most of this was being burned countering the search space increasing as irreversible operations had multiple possible inputs.

At this point I found my first result, which I could have found earlier if I'd just let things run longer. One of the usernames only had seven characters: "Procyon". However, I was still hitting a massive time investment wall. Checking all those constraints took time.

Then I realized the 50% reduction in search space from the constraints was wrong. It turned out that the constraints were just catching what would have been caught by the very next reverse-multiplication or reverse-division-by-3. The constraints were actually achieving a... 0% reduction. Whoops. Removing them sped things up quite a bit, allowing me to search all 9 character strings.

Finally, I realized that I should switch the direction of caching. Going backwards was more expensive than going forwards, and I was memory-limited to caching fewer rounds than I was exploring from the other direction. Caching the results of going backwards, instead of going forwards, would reduce the amount of reverse hash operations and allow me to search all strings up to ten characters as long as I was willing to wait a couple days while my laptop chugged away.

Collision

We've finally reached the weakness I ultimately used to beat the hash function: the size of its output.

The output size is 64 bits, which allows a bit more than 10^{19} possibilities. I can search through every string up to ten characters (with 93 possibilities per character), which is 93^{10} possibilities. That's about five times 10^{19}.

Right. At this point it doesn't matter how long the real password is. By pure brute luck, I'm going to stumble onto strings that hash to the same thing.

My work is done. I just need to let the computer churn.

Code

This is the code I used to break the hash function:

```
/// Returns a given start state and a sequence of values of the given
/// length that reach the given end state. If not such sequence
/// exists, returns null.
public static Tuple<HashState, int[]> Break(HashState end,
                                            int assumedLength,
                                            IEnumerable<HashState>
startStates) {
    // generate bloom filter going backwards from end
    var numExpandBackward = (assumedLength - 1).Min((assumedLength *
2) / 3).Max(0).Min(4);
    var filter = HashStateBloomFilter.GenReverseCache(end, numExpand-
Backward, pFalsePositive: 0.0001);

    // explore forward from starts to filter, discard states that
    // don't match
    var possiblePartialSolutions =
        from start in startStates
        from midStateAndData in start.ExploreTraceVolatile(assumedLen
gth - numExpandBackward)
        where filter.MayContain(midStateAndData.Item1)
        select new { start, data = midStateAndData.Item2.ToArray(),
end = midStateAndData.Item1 };

    // base case: not enough length to bother meeting in the middle.
    // Partials are actually complete solutions.
    if (numExpandBackward == 0) {
        return possiblePartialSolutions
            .Select(e => Tuple.Create(e.start, e.data))
            .FirstOrDefault();
    }
    // we don't want to wait for all possible partial solutions before
    // checking. That would take tons of memory. we also don't want
    // to check after every single possible partial solution,
    // because that's expensive. so we partition possible solutions
    // and check whenever there's enough to make it worth the time.
```

```
    var partitions = possiblePartialSolutions.PartitionVolatile(10000);

    // complete any partial solutions
    var solutions =
        from partition in partitions
        let partialSolutionMap = partition.ToDictionary(e => e.end, e
=> e)
        // recursively solve the gap
        let secondHalf = Break(end, numExpandBackward, partialSolu-
tionMap.Keys, true)
        where secondHalf != null
        // Anything reaching here is a solution. Combine it with the
        // first half and return it.
        let partialSolution = partialSolutionMap[secondHalf.Item1]
        let start = partialSolution.start
        let data = partialSolution.data.Concat(secondHalf.Item2).
ToArray()
        select Tuple.Create(start, data);

    // actually run the queries
    return solutions.FirstOrDefault();
}
```

The above code makes a bloom filter containing states that can reach the end by adding a suffix of some length (up to 4). It then iterates over prefixes of the complementary length, noting any that match the filter. Once it has ten thousand matching prefixes, or runs out, it recursively tries to break the gap from the states reached by matching prefixes to the end state. If it finds a way to break the gap, the correct prefix is paired with the gap solution in order to make a full solution. Otherwise it keeps going until it runs out of prefixes.

Note that the code is not optimized very much. In particular, it's using Linq queries instead of the equivalent imperative code. As far as I know, neither the C# compiler not the .Net jit optimize them particularly well, and so the code is paying for tons of virtual function calls when it doesn't have to. On the other hand, the equivalent imperative code is stupidly hard to get right because you end up mixing everything together in a big jumble. (I spent my time doing other things while the computer did the tedious work.)

Solutions

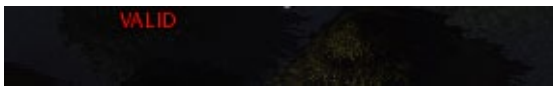
After about two days of computing, and one dead laptop, the code returned a password that matched the password hash. The password is "<+nt1Akg-bMht" (or rather, <+nt1AkgbMht is a string that hashes to the same thing as the true password). If you're wondering why the password has 12 characters, when I said I was searching 10, recall that the first two characters of the password were given away in the JASS code. I searched 10 additional characters.

(It's tempting to pretend I didn't know those two characters, because $93^{12} \approx 10^{24}$, so I could say I literally searched a trillion trillion possibilities.)

After another three days, I had both remaining usernames. These are clearly collisions, instead of the actual names, but here they are nonetheless: "hRlGz%W3&R" and "b>4FXV'Xf8" match the first and third hashed usernames respectively (the second was "Procyon").

My Reward

With the solutions in hand I can finally download Phase Killer, play it in single player with a profile called "Procyon", say "<+nt1AkgbMht" and see... a "VALID" message.



Worth it.

Summary

Things we've learned about writing hash functions:

- Don't write your own hash function.
- Don't leak entropy. All round operations should be reversible.
- Don't use the hash's entire state as its result. Running backwards from the result should be hard. (See also: length extension attack.)
- Use non-linear combinations of operations and apply them a lot. The effects of each input should be difficult to separate. (See also: avalanche effect.)
- Have a result with lots of bits. Collisions should be hard to find. (See also: birthday attack.)
- Don't write your own hash function (except for fun). ■

Craig Gidney is a computer scientist who started his adventure making maps and mods for Starcraft and Warcraft III. He works at Twisted Oak Studios, a software consulting cooperative in Halifax, Nova Scotia. A software developer to the core, he enjoys coding both personally and professionally, in theory and in practice.

Reprinted with permission of the original author.
First appeared in *hn.my/toyhash* (twistedoakstudios.com)

HOW TO LEARN EMACS

STEP ∅ (for geeks)

If you're a developer or sysad...

Learn Vim → the other text editor

It's okay. Learn the basics so that you can easily work on other people's computers. If you know your way around Vim, people won't give you as much grief over Emacs.

Bonus mini-cheatsheet!

Here's what you need to know in Vi:

i insert mode ↔ <Esc> command mode

Long-time Vi user trying out Emacs?

- :vimtutor
- :w write/save file
- :q quit
- :q! really quit

evil mode Emacs

You will probably meet editor zealots.

Be ambitextrous!

You can actually edit remote files in Emacs, but that's an intermediate topic. (Curious? See TRAMP)

Okay. Once you know the basics of Vim, you can get on with learning Emacs. ☺

Why Emacs?

- Amazingly customizable
- Endless room for growth

Most people start here!

STEP 1

Learn how to learn

There are some old books on Emacs, but the version differences can be rather confusing.

Start with the built-in tutorial instead.

Help → Emacs Tutorial ← repeat as many times as you need to

Other resources:

- emacswiki.org — Lots of resources
- planet.emacsen.org — Emacs-related blogs
- IRC: irc.freenode.net → #emacs — Great for help and hanging out

Can't use the menu? Press Control-h t to start.

NOTE: The Emacs tutorial has lots of weird terms: "Meta key," "frame," "buffer." This is because Emacs started a *long* time ago. Don't worry, you'll get the hang of it with practice.

Some things that might help:

C-x C-s: This is how keyboard shortcuts are written.

① press Control & x at the same time

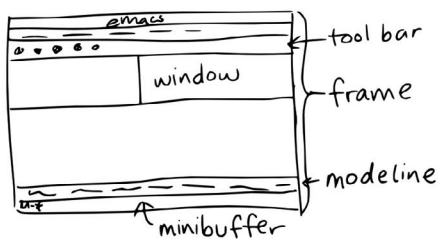
② then let go of x and press Control & s

Tip: Since you're using Control for both keys, you can hold Control down instead of letting go between x and s.

M-x lets you call commands by name, which is great if you can't remember the keyboard shortcut. Ex: M-x help-with-tutorial RET

M-x → Meta (usually Alt- or ⌘-)

RET → This means press Enter/Return



Other good help commands:

- C-h i manual
- C-h k <keyboard shortcut> gives you help on it
- C-h f describes commands/functions
- C-h a searches for commands
- C-h C-h shows help

Windows show buffers, which could be

- files
- processes
- other info

A beginner's guide to Emacs 24 or later
Sacha Chua (@sachac) · sachachua.com/begin-emacs

Questions? I'd love to hear from you!

2013(v2)

STEP 2 Learn Emacs basics

C-x C-f open (find-file)
C-x C-s save
C-x C-c quit

NOTE: You don't need to quit Emacs after each file. Just leave it running and use C-x C-f to open the next.

How to select text

← Go to the start of your selection and press C-SPC (Control + Space)
Go to the end of your selection and run your command

C-w cut (kill)
C-y paste (yank)
↳ M-y paste older things
C-/ undo

*Learn how to use Keyboard macros. They're awesome.
C-x (start macro
C-x) end macro
C-x e execute macro
e ... ...again

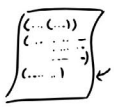
STEP 3 Extend & customize

M-x load-theme RET
Try out color themes → use only to see the list and list-packages to add more
M-x customize-group RET
set common options
M-x customize-face RET
change background, foreground, etc.

M-x list-packages RET
install lots of modules... → Want a prettier Emacs? Check out color themes! (Lots of people like color-theme-solarized)

and then...

editing your ~/.emacs.d/init.el file!
↳ Huh? Where? Just use C-x C-f to create it!



initializes your Emacs, adds new functionality, and so on.

See emacswiki.org for lots of examples

Use M-x eval-buffer or restart Emacs to see the changes.

Broke your Emacs config? } emacs -q skips your customizations

STEP 4 Learn other handy tips

Buffer & window management

C-x b switch buffer → even better with M-x ido-mode
C-x 2 split
C-x 3 split
C-x o other window
C-x 0 get rid of current window
C-x 1 get rid of other windows

Navigation & search

M-g M-g go to line (hold Alt/⌘ and then press g twice)
C-s Interactive search
C-r Interactive search backward
M-> End of buffer
M-< Beginning of buffer
M-x occur RET Find lines

STEP 5 Explore!

Org-mode.org
organize your life in plain text

Narrowing/Widening focus.

TRAMP remote access

Eshell / Term command-line in Emacs

Calc powerful calculator and converter

((·)) Writing & debugging Emacs Lisp (edebug)
(it sounds scary, but it's powerful ☺)

There's so much more!

Ask away, and discover more by exploring!

Sacha Chua

Nginx for Developers: An Introduction

By KYLE MACDONALD & JEFF ESCALANTE

If you are a web developer, you've probably heard of nginx (pronounced engine-x). Nginx is a fast and extremely powerful http and reverse proxy server that can be used to quickly and easily serve webpages.

Unfortunately, like many sysops tools, there is very little documentation and very few tutorials that explain how it works and how to get it up and running. There is a wiki, which is extensive and confusing, showing you all possible options rather than presenting the important ones as you need them. After struggling with it myself for a bit, I finally got down the basics of how to work with nginx, and wanted to share it so that other developers would have an easier time picking it up.

So let's dive right into it. For this tutorial, you're going to want a VPS of some sort, preferably fresh so that you can avoid potential conflict with other old setups etc.

Initial Setup

Assuming you are running an Ubuntu box, once you have set up your login and have `apt` updated, just run `apt-get install nginx` and everything should install cleanly. Visit your server's IP address in a web browser and you'll see the "welcome to nginx" message. Great success.

Finding nginx

When nginx is installed (through `apt`), it provides a solid basic structure for how to set up your config files. All nginx config files are located in `/etc/nginx`, so `cd` there and poke around. The place you'll want to add new configurations is the `sites-enabled` folder. If you check this folder out, you'll find that there's a single text file called `default` in there, and opening that up you'll see an nginx configuration and the code that causes the "welcome to nginx" page to display. Now let's make

our own config file with the bare basics to display a page. Touch a new file inside sites-enabled called "test", open it up in your text editor of choice, and let's get to it.

Note: You'll also find a `/etc/nginx/sites-available` directory. If you find yourself managing many different sites that are coming up-and-down, this folder can help keep things organized. Add your nginx configuration files here instead and then symlink them to `sites-enabled`. This command might look something like this...

```
ln -s /etc/nginx/sites-available/
dotcom /etc/nginx/sites-enabled/
dotcom
```

Only configurations in `sites-enabled` will actually be public to visitors, but you may want to keep some configurations in `sites-available` for archival and symlinking purposes.

Configuring a Static Server

Nginx config files use their own language, but the good news is that it's super simple. Much like css, namespaces are declared followed by a block which is bound on either side by curly braces. The top level block we want to enter is server, which would look like this:

```
server {

}
```

Inside this block, we can, again much like css, add key-value pairs followed by semicolons, or (more like sass), we can add a nested block. We'll be doing both of these for the basic setup, and it should be no problem to follow.

There is a ridiculous amount of possible key-value pairs or blocks we can add (called directives; I'll be referring to them this way through the rest of the tutorial), and if you jump into the documentation [wiki.nginx.org/DirectiveIndex], you will find a few hundred. For a basic server setup though, only a few are important to know, and we'll go over those here. I'll include links to the official nginx docs for each directive we go over if you are interested. Since the official docs are the only official way to get around nginx, it's important to become familiar with how they work if and when you want to set up something more advanced later.

`listen`

This directive specifies the port that your server will listen at. If you have ever worked with rails, you'd know that the local server runs on port 3000. Roots runs on port 1111. SSL runs on port 443. The default port for the internet is 80, so if there's no port in a url, that means it's 80. Since you are likely trying to run a production server here, it's likely that you will be after port 80, so let's enter that here.

```
server {
  listen 80;
}
```

Note that this is the default and is not strictly necessary to enter, but it's good to do it anyway in this case to show what's going on. Bam, first directive written. Feels great. Let's keep rolling.

`server_name`

This directive is essentially a matcher for the url bar. Whenever any sort of request comes in to nginx, it takes a look at the url and looks for a `server` block that has a matching `server_name` directive. So if your site was at http://example.com, your `server_name` for the root would be example.com. If you used an A Record to also route http://snargles.com through to your server, you could add another server block with a `server_name` of snargles.com, and that block would match requests coming in from that domain.

This is quite powerful. If you think about it, this means you can host numerous sites, even coming from different domains, on a single nginx configuration. All you have to do is set up an A Record that points the domain to your box's IP, then sort out the rest with nginx server configs.

It's worth noting two more interesting aspects of `server_name`. First, you can use this directive to also deal with subdomains. If you want to match http://test.example.com, you can easily do this, and even map it to an entirely different app. Second, you can perform some wizardry with the value of `server_name`. You can use both wildcards, indicated by *, or regular expressions to match routes. As you can imagine, this can be extremely powerful. Let's write a quick config for the root domain of example.com.

```
server {
  listen 80;
  server_name example.com;
}
```

Sweet. Only a couple more directives till we can get our site in production.

`root`

This is the key to serving static sites. If you are just trying to lay down some html and css, the root directive specifies the directory that you have stored your files in. I like to store my sites in `/var/www`, so let's go make a folder there. Just `mkdir` a folder called `/var/www/example`, and inside this, touch an `index.html` file and add a paragraph saying "hello world" or something. Now that we're good, let's get back to our config and add our new document root:

```
server {
  listen 80;
  server_name example.com;
  root /var/www/example;
}
```

Now that we've got the basic variables set, let's actually listen for hits to a specific route.

`location`
Location takes two parameters, a string/regex and a block. The string/regex is a matcher for a specific location. So if you wanted anyone who went to example.com/whatever to hit a specific page, you would use "whatever" as the uri. In this case, we are just trying to match the root, so we can use / as the uri here. Let's fill in an empty block for now, which we will complete in a second.

```
server {
  listen 80;
  server_name example.com;
  root /var/www/example;

  location / {

  }
}
```

Note that the first parameter has a number of options which you can see in the linked documentation, and that the ability to match by regex is quite powerful. Inside that block, we want to actually route to the result page. Also note that this / uri will match all urls, since it's treated as a regex. If you want a location block to match only an exact string, you can preface it with an equals sign, as shown below. But in this case, it's ok for it to match all urls.

```
location = / { ... }
```

Now to fill in that block from before.... We can use another directive inside the block to serve a file called `try_files`. Try files takes a list of filenames or patterns that it will try to find in you root directory, and it will serve the first one it finds. For our simple static server, we want to try to find a file with the name of "whatever" after the slash, like "whatever.html". If there is nothing after the slash, it should go for index.html. There are a few other technical aspects to how you write these which are laid out in the docs linked above; here's a very simple implementation.

```
server {
  listen 80;
  server_name example.com;
  root /var/www/example;

  location / {
    try_files $uri $uri/ /index.
html;
  }
}
```

Now you might ask yourself, where did this `$uri` business come from? Well, that's the magic of nginx. As soon as that request comes in, nginx makes a bunch of variables available to you that hold information about the request. In this case `uri` is exactly what we were after. So let's walk through the list.

- Request comes in for http://example.com, nginx fields it.
- nginx finds the server block with a `server_name` of example.com, and picks this one to handle the request.
- nginx matches the request against any location blocks present. Since the / block matches anything after the root domain, we have a match.
- Inside the matching location block, nginx decides to try serving a file up. First, it looks for a file named nothing, since uri maps to nothing in this case — no luck. Then it looks for a directory called nothing, still no luck. Finally, it looks for /index.html within the root directory, `/var/www/example`, and finds that file. It then serves it up to you.

Now try to imagine the flow if we were to add a `test.html` file to the root directory and go to http://example.com/test.html. Then try it and see what happens.

Note that you can twist this configuration any way you want. For example, on carrot.is, we have it configured so that when you hit a filename *without* the .html extension, `try_files` looks for `$uri.html` and matches that as well. So you could go to http://carrot.is/about as well as http://carrot.is/about.html, and they would both return the same document. The amount of wizardry you can do with the server config is limited only to your crazy ambitions.

Ship It

Okay, so what have we actually done here? What we've done is that we've added a `server` declaration to nginx's configuration. When nginx runs, it slurps up all of the configuration files you have put into `/etc/sites-enabled` and uses those to know what to display to your viewers. But wait! If you stopped here, you might not be able to see your new server — this is because nginx doesn't quite know about your new changes yet. To get nginx to know about your new configuration you need to reload nginx so that it can pull in your new configuration. The easiest way to do this is to run

```
service nginx reload
```

Note: This `service` command actually just aliases to running the `reload` command on the configuration files that `apt` installed into your server's file system. In this example it aliases to `/etc/init.d/nginx reload`.

Another thing you may want to do is test your configurations to assure they are valid. To do this simply run, `service nginx -t`.

Then just visit your server's IP address again and you should see your shiny new page! ■

Kyle MacDonald — Chief Technology Officer and Partner at Carrot Creative. Kyle drives all technology decisions and leads development on client projects — crafting an end-to-end experience for users and translating that into code. Kyle has successfully launched digital projects for clients including: Red Bull, Target, Disney, Crayola, The Home Depot, Ford, Budweiser and more.

Jeff Escalante — With a strong background in neuroscience, user experience, graphic design, and programming, Jeff is a Developer at Carrot. He made roots [roots.cx] — integral to the Carrot site [carrot.is], and is a front-end specialist. For more see, github.com/jenius

Carrot is a full-service digital agency headquartered in Brooklyn, NY.

Reprinted with permission of the original author.
First appeared in *hn.my/nginxintro* (carrot.is)

The Unprofitable SaaS Business Model Trap

By JASON COHEN

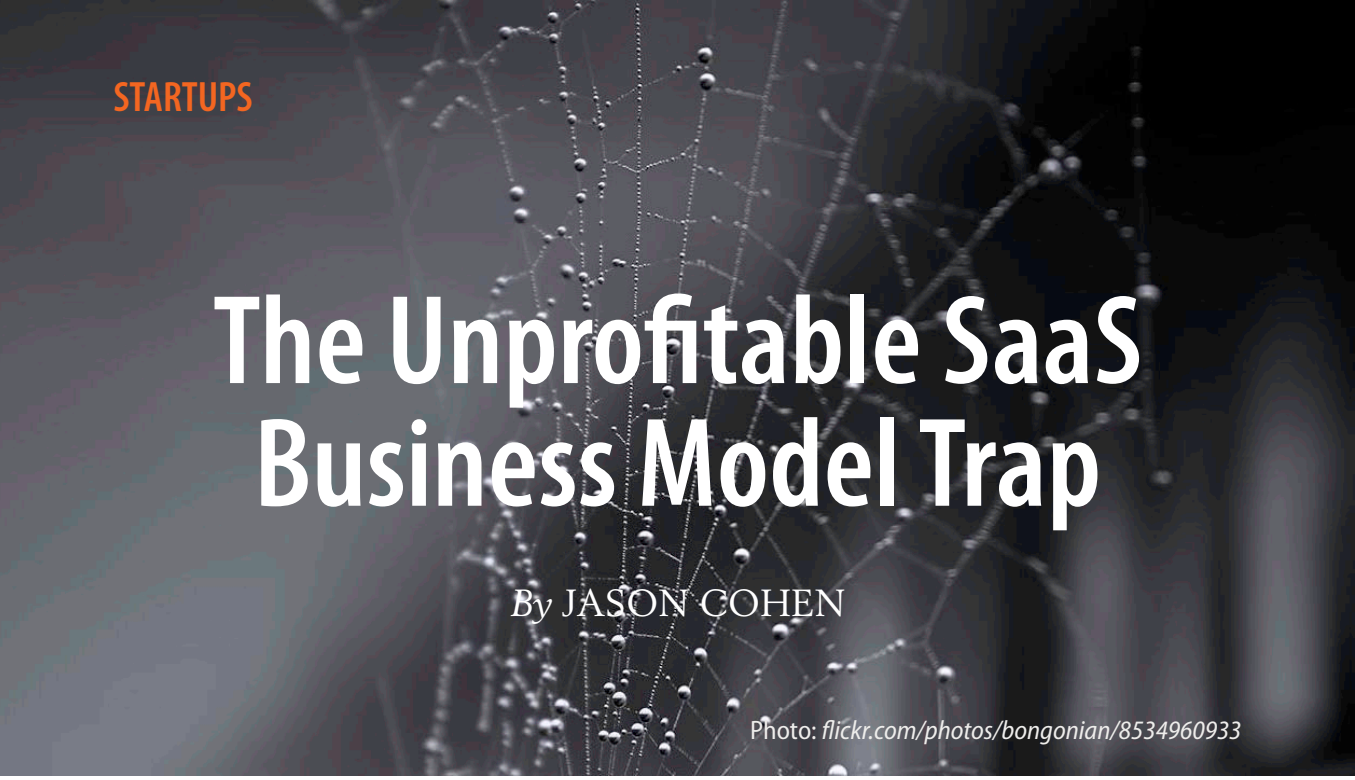


Photo: *flickr.com/photos/bongonian/8534960933*

Marketo filed for IPO with impressive 80% year-over-year growth in 2012 and almost $60M in revenue.

Except, they lost $35M. WTF?

It's not impressive when you spend $1.60 for every $1.00 of revenue, force-feeding sales pipelines with an unprofitable product.

Don't tell me this is normal for growing enterprise SaaS companies. I know the argument: The pay-back period on sales, marketing, and up-start costs is long, but there's a profitable result at the end of the tunnel. Just wait!

Bullshit. Eloqua was also a SaaS company, also selling to enterprise and selling the same product in exactly the same space. It was also tightly integrated with Salesforce.com, and IPO'ed with a $5M loss on $71M in revenue — a 7% loss instead of Marketo's massive 60% loss.

So no, this upside-down business model isn't what a SaaS business should construct. I wish the modern startup community would understand the mindset that gets a company to this point, and resist it.

The mindset works like this:

1. It costs a lot of money to land an enterprise customer: marketing, sales, legal, account management, on-boarding, technical guidance, training, etc. And how many times do you run through that process and still lose the customer? So these costs are amortized over the customers you do land.

2. SaaS companies earn their revenue over time. Whereas a normal software company might charge $100,000 for an enterprise deal, and thus immediately earn back those "customer startup" costs plus profit, the same SaaS deal might be $5,000/month, and it might take 18 months to get that same amount of revenue. The good news is that after those 18 months, the SaaS company still charges $5,000/mo. The other company has to bust ass for measly 20%/year maintenance fees.
3. As a result, enterprise-facing SaaS companies are unprofitable for the first 12-24 months of a given customer's life.
4. But, a growing SaaS company will be landing new customers and increasing numbers, which means piling up more and more unprofitable operations.
5. So much so, that even when an older customer individually crosses into profitability, there are so many more unprofitable customers, the company remains permanently unprofitable so long as it maintains healthy growth.
6. Plus, there's all the other costs — R&D to build the stuff, office space, executive salaries, billing, legal, finance, HR, tech support, account managers, etc. To actually be profitable, you need to cover those costs too. So it takes even longer to be bottom-line-profitable.
7. Therefore, it is healthy and reasonable for SaaS companies to be unprofitable as long as they're growing even a little bit.

Early in a company's life, this line of reasoning is correct. But at Marketo's size, this argument falls apart.

Why, exactly?

There's a tacit assumption that if only we just stopped spending to grow, we'd be profitable. Thus, this "really is" a profitable company, and the only reason it's not is because of growth, which means market domination, which is a Good Thing.

The fallacy is: That time never comes. No company stops trying to grow! The mythical time when growth rates are small so the company reaps the rewards of having a huge stable of profitable customers never arrives. When do you "show me the money?"

It's worse. Growth becomes harder and harder for SaaS companies because of cancellations. Even with a great retention rate (e.g., 75%/year), you have to replace 25% of your revenue with new — which means unprofitable — customers just to break even in top-line revenue! More losses, more unprofitability.

Even with very broad numbers, you can see how this model doesn't work. Here's typical numbers for an enterprise SaaS company at scale:

- 1.5 year pay-back period (i.e., time to earn back the revenue to cover all your customer acquisition expenses).
- 75% annual retention (which also means you turn over the entire customer base every 4 years. On average, of course — some stay longer, many shorter).
- 30% cost to serve the customer (which can also be stated at 70% Gross Profit Margin, meaning for every $1.00 of revenue, $0.30 disappears in direct costs to service that customer, like servers, licenses, tech support, and account management. Many public SaaS companies, even the titans like Salesforce.com, are about 70% GPM).
- 15% revenue == cost for R&D department.
- 15% revenue == cost for Admin department (office space, finance, HR, execs).

Say the average customer represents R dollars in annual revenue. That's:

- $4R of revenue over the lifetime of the customer. But:
- $1.5R is spent to acquire the customer (the pay-back period).
- $1.2R is spent in gross margin to service the customer (4 years times 30% cost).
- $0.6R spent on R&D (15% over 4 years).
- $0.6R spent on Admin (15% over 4 years).

So out of the original $4R, we're left with $0.1R in profit. That's 1/40th of the revenue making its way to actual bottom-line profitability, and even that takes 4 years to achieve.

And that is without any growth at all. But you need to grow enough to keep up with cancellations at minimum, so that consumes the last notion of profitability.

What's the solution?

Successful, profitable SaaS companies at scale (certainly by $30M/year revenue, but should be paying attention to this stuff by $5M/year), do several things to make the math work:

- Undo the effect of cancellations through up-sells/upgrades. Salesforce.com and ZenDesk charge more for every person you add, and more per person when you increase the features in your plan. Their customers grow (on average). Thus, their revenue over four years is not 4R, but rather it might be R on the first year, 1.5R on the second, 2R on the third, etc., so perhaps 7R in four years. That drastically changes the equation, because cost to "acquire" the customer doesn't go up, and in general R&D and Admin don't either. Taking "rate of cancellations" minus "rate of upgrades" is called "net churn." Getting to zero net-churn is a big step in getting profitable; the most successful SaaS companies have negative net churn. It's not just pure software companies that achieve this — hardware/server SaaS company Rackspace also has negative net churn, which enables them to grow revenues 30% year-over-year with $1.5B in revenue and $300M in profit.
- Use viral growth to offset cancellations. Few B2B companies can truly claim "viral growth" characteristics. But for the few who do, they can maintain growth rates of X%/year where X is much larger than cancellation, and do so with very little acquisition costs. In this case, cancellation never "catches up," and you win.
- Drastically reduce the cost of customer acquisition. An 18-month payback period is a killer. If customers can be found with paid advertising; if they can sign up without talking to a sales person; if they can learn the product through in-product tutorials, great documentation, and how-to videos; if they can import their data without assistance; if they can demonstrate value to the purse-string-holders without a sales person writing the presentation for them, then the cost of cancellation-replacement and proper growth becomes small enough that it's no longer a barrier to profitability, even under conditions of growth.

- Drastically improve GPM. It's hard for a service-oriented enterprise-sales company to not have real costs around tech support, account management, and extensive IT infrastructure, which is why even the most cost-efficient (and profitable!) enterprise-facing SaaS companies often can't push much past 70% GPM (e.g. Salesforce.com, Rackspace). But, companies with extremely low-touch customer service (which doesn't necessarily mean bad customer service!) can push it way up (Google, Facebook, Freshbooks), unlocking "free money" for profitability.

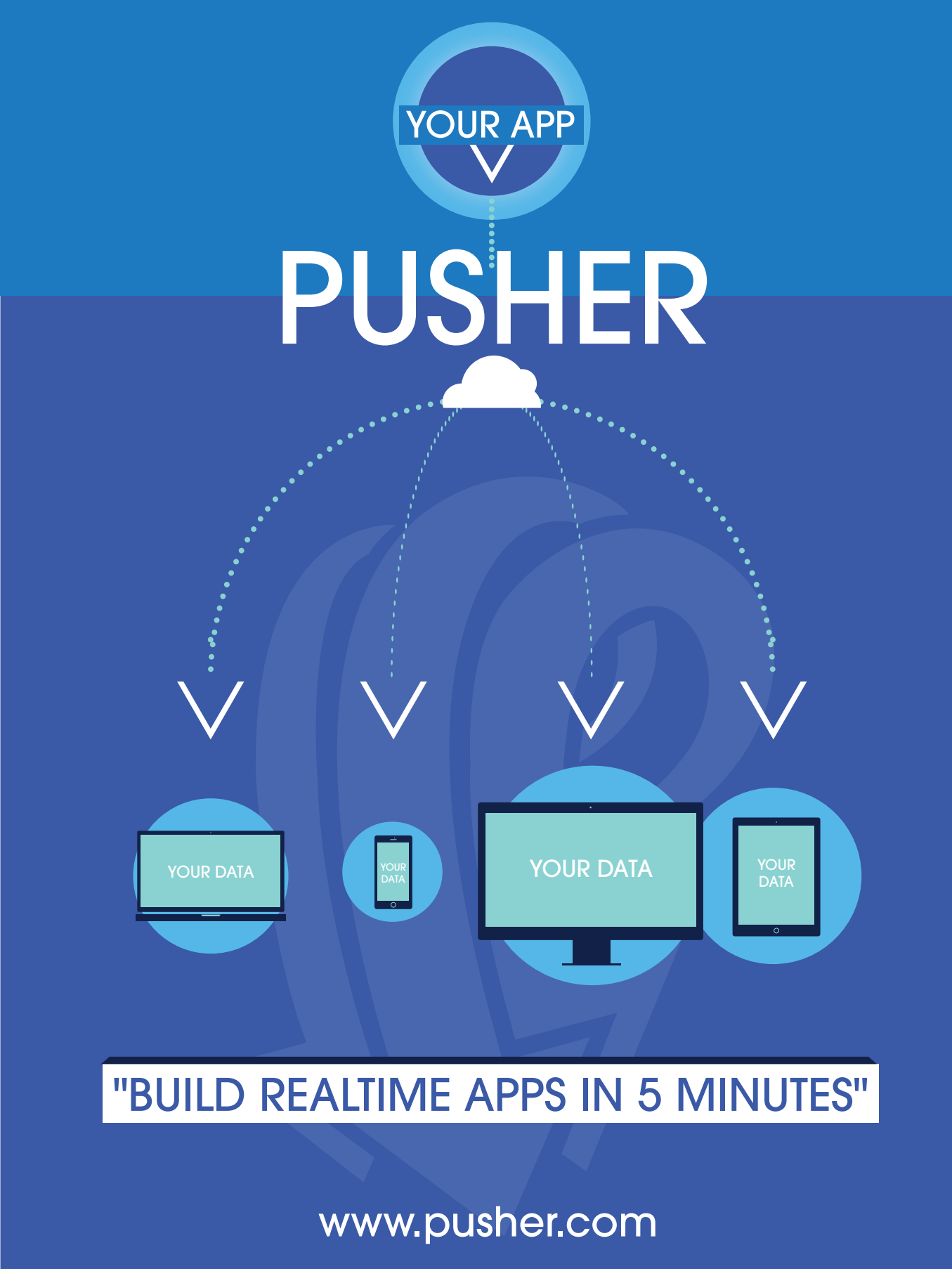
Another way to think about these solutions is that a SaaS business cannot have static fundamental metrics. The metrics themselves need to improve — lowering cancellation rates, lowering net churn, increasing GPM, reducing cost to acquire customers. Leaving the metrics alone, and trying to "grow until profitable" doesn't work.

It's like the old Jackie Mason joke — A man is selling jackets at cost. The customer asks "how can you sell at cost, how do you make any money?" Answer: "I sell a lot of jackets!"

Marketo is selling a lot of jackets. ■

Jason Cohen is the founder of WP Engine — Heroku for WordPress, after exitting from three previous companies. He blogs at *blog.asmartbear.com*

Reprinted with permission of the original author.
First appeared in *hn.my/saas* (asmartbear.com)


YOUR APP
PUSHER
YOUR DATA
YOUR DATA
YOUR DATA
YOUR DATA
"BUILD REALTIME APPS IN 5 MINUTES"
www.pusher.com

Procrastination Is Not Laziness

By DAVID CAIN

I WAS GOING TO tackle my procrastination problem last weekend, but I never got around to it.

By Sunday at 5:48 p.m., I realized I had blown it again. Throughout the week I feel like I barely have enough time to cook, eat, tidy up, write an article and do the odd errand. I lean towards the weekend, when I have two whole days to finally get some work done. To improve my blog, to catch up on my correspondence, to get some monkeys off my back like fixing things that need fixing, organizing things that need organizing, tackling things that need tackling.

But the weekends go by and I never catch up. I don't use the time well. Time is not what I'm short on, even though that's what I tell myself all week.

Sometimes I do sit down early in the day and pound something out, but then I give myself a well-deserved break and that's usually the end of any productivity. I end up clicking around on the internet, then clean up, then cook something, then watch a bit of a documentary online, then try to work again, then get distracted. Then I decide to wait until after supper to do some work, then I start reading something after supper, then if I'm still home, it's already after 9:00 so I decide I'll get an early start the next day.

I avoid taking on the real important stuff. I create work of secondary importance so that I never have to confront the really worthwhile things. When I get on a roll, I back off and stay backed off. I take breaks that turn into written-off days. I am addicted to hanging it up for the night, to letting myself off the hook.

The important stuff doesn't get done, at least not before my procrastinatory tendencies have created an obvious, impending consequence of not doing it, like incurring a fine, really letting someone down, or getting fired.

So much of what I want to do isn't terribly difficult and wouldn't take a lot of time to get done. Looking at my projects list now I have items like: book an appointment for X, send in that change of address form, phone so-and-so about Y, write a short piece for Z. And many of them have been sitting there for weeks or months. I have the most bizarre aversion to tackling things.

Reaching critical levels

To some of you this is already sounding familiar.

I have lived with this sort of "productivity lag" most of my life, but it only recently hit me that it's not just run-of-the-mill human busyness. Some alarming patterns have emerged in the past few months. I've been feeling chronic stress for the first time in years. I have been waking up angry on a fairly regular basis, and that's not okay.

After a bit of poking around at the library, it's become clear to me that I have a pretty serious procrastination issue. I also learned that procrastination is not caused by laziness or disorganization, but by deeper psychological issues, which I'll touch on a bit later in this post.

As I said, it's always been a feature of my life but it's reached a critical point this year. The catalyst has been a change in my job. At the end of January I was dropped into a new role that I neither like nor feel prepared for. My protests were met with, "You'll figure it out as you go along; it's like this for everyone at first." I have since worked it through, mostly, but not before it set off a pretty bad stress cycle that brought some ugly stuff to the surface.

Honestly, it probably would have been a much easier adjustment for most people to make than it has been for me, but my initial uncertainty combined in very ugly ways with my lifelong phobias of asking for help, admitting ignorance, and talking to people I don't know on the phone. Paralysis set in. Stress, which has been a mostly-dormant force in my life for the last five years or so, became prominent again.

Once you lose track of the specific items that are causing you stress, you tend to regard it all as one big ugly entity that you want to avoid. My unaddressed duties and grey areas at work became mixed with my unaddressed duties and grey areas outside of work, to create a stifling mutant stressor that only leaves me alone while I'm sleeping. All the work I've done towards learning to effect the quality of the day can be easily short-circuited by my procrastination habit, and that's what's happening right now. It has gone way too far and I am determined to address the bad habits that let it get this way.

My last few experiments have created huge changes in the way I operate and the environment I live in. Well I'm

doing a bigger one this time. I'm taking on a problem that has probably taken more from me than any other behavior. I've lost so many opportunities, relationships, advantages, sources of income and growth. There is certainly nothing that has caused more suffering in my life than my propensity to avoid achievement or competition.

For what I'm capable of, I have been a resoundingly unproductive person. Almost every Sunday night I mourn another blown opportunity to catch up, and throughout every week I am leaning towards the next weekend. The weeks fly by, and if weeks are flying by, so are months. How we spend our days is how we spend our lives, and I've had enough of this.

Monday I'll formally announce Experiment No. 11. While preparing for it I did some research on where procrastination comes from, which was frankly quite alarming to me and shed a sorely-needed light on why I have had such confounding, persistent trouble with getting ordinary things done. This post is quite a bit longer than usual but if you've had similar trouble, it might just shake loose something that's been stuck for a very long time.

The real causes of procrastination

Let's clear something up: I am not lazy. I have no shortage of energy, I have no interest in lounging on the couch, I don't have TV service, I never wear pajamas all day. Waking up after 7:30 is sleeping in for me, even on a Saturday. I actually like working.

Yet I exhibit a consistent failure to work through my day-to-day tasks, errands and projects in any manner than could be considered timely. Nearly everything must reach some sort of "scary point" for me to finally move on it. Like when I waited till the last possible day to submit my lease renewal, even though I had three months of lead time. In the end it took about fifteen minutes, but evidently I needed to be a day away from losing my home in order to do it.

I ended up reading one of the more highly acclaimed books on procrastination, Neil Fiore's The Now Habit. Reading the section on the psychological causes of procrastination really hit home.

It turns out procrastination is not typically a function of laziness, apathy or work ethic as it is often regarded to be. It's a neurotic self-defense behavior that develops to protect a person's sense of self-worth.

You see, procrastinators tend to be people who have, for whatever reason, developed to perceive an unusually strong association between their

performance and their value as a person. This makes failure or criticism disproportionately painful, which leads naturally to hesitancy when it comes to the prospect of doing anything that reflects their ability — which is pretty much everything.

But in real life, you can't avoid doing things. We have to earn a living, do our taxes, have difficult conversations sometimes. Human life requires confronting uncertainty and risk, so pressure mounts. Procrastination gives a person a temporary hit of relief from this pressure of "having to do" things, which is a self-rewarding behavior. So it continues and becomes the normal way to respond to these pressures.

Particularly prone to serious procrastination problems are children who grew up with unusually high expectations placed on them. Their older siblings may have been high achievers, leaving big shoes to fill, or their parents may have had neurotic and inhuman expectations of their own, or else they exhibited exceptional talents early on, and thereafter "average" performances were met with concern and suspicion from parents and teachers.

This was the part that made my heart sink when I read it. Not that anybody was trying to make things difficult for me, but I grew up feeling high expectations from the adults in my life and myself. For most of my schooling, I was always in advanced programs, always aced everything, and when I got anything less than an A, people asked me what was wrong.

I also noticed other kids didn't get this treatment. They were congratulated for getting Bs and even Cs. So from the feedback I got, I learned that a report card (of mine) with five As and a B was indicative of a shortcoming somewhere, not success. I've written about this before so I won't get into it here, but suffice it to say that I learned that the downsides of being imperfect are far greater than the upsides of being perfect.

Perfectionism breeds pessimism

It was a major revelation to me when I recognized a year ago that despite my preference for and sensitivity to the positive aspects of life, I am a pessimist — I have come to give potential downsides far more weight than potential upsides. This means that pushing projects ahead is — on the balance — a bad deal, because unless I'm pretty damn perfect there is much more pain to be had in doing that than pleasure.

This is obviously an inaccurate presumption, and I'm intellectually aware of that, but when it comes down to confronting it "in the field," it's amazing how tricky the mind can be. I have a lifetime of habits routing me away from striving for prizes in life, and towards protecting myself.

For a procrastinator of my kind, perfection (or something negligibly close to it) thereby becomes the only result that allows one to be comfortable with himself. A procrastinator becomes disproportionately motivated by the pain of failure. So when you consider taking anything on, the promise of praise or benefit from doing something right are overshadowed by the (disproportionately greater) threat of getting something wrong. Growing up under such high expectations, people learn to associate imperfection or criticism with outright failure, and failure with personal inadequacy.

A person who does not have this neurosis might wish they didn't make a mistake, whereas the neurotic procrastinator perceives the error as being a reflection of their character. In other words, most people suffer mainly the practical consequences of mistakes (such as finishing with a lower grade, or having to redo something) with only minor self-esteem implications, while neurotic procrastinators perceive every mistake they make as being a flaw in them.

So what they are motivated to do is to avoid finishing anything, because to complete and submit work is to subject yourself (not just your work) to scrutiny. To move forward with any task is to subject yourself to risks that appear to the subconscious to be positively deadly because part of you is convinced that it is you that is at stake, not just your time, resources, patience, options or other secondary considerations. To the fear center of your brain, by acting without guarantees of success (and there are none), you really are facing annihilation.

A backlog of avoided tasks accumulates, and each one represents another series of threats to your self-worth should you tackle them. So the fear mounts, knowing that there is a minefield of threats between you and the fulfillment of your responsibilities. You feel like you must do something and can't do that thing simultaneously, which can only lead to a burning resentment of the people or forces that put you in that impossible place — your employer, your society, or yourself. A victim mentality emerges.

Because it is rewarding on the short term, procrastination eventually takes on the form of an addiction to the temporary relief from these deep-rooted fears. Procrastinators get an extremely gratifying "hit" whenever they decide to let themselves off the hook for the rest of the day, only to wake up to a more tightly squeezed day with even less confidence.

Once a pattern of procrastination is established, it can be perpetuated for reasons other than the fear of failure. For example, if you know you have a track record of taking weeks to finally do something that might only take

two hours if you weren't averse to it, you begin to see every non-simple task as a potentially endless struggle. So a modest list of 10-12 medium-complexity to-dos might represent to you an insurmountable amount of work, so it feels hopeless just to start one little part of one task. This hones a hair-trigger overwhelm response, and life gets really difficult really easily.

All I want

As I mentioned, on Monday I will begin Experiment 11, which is direct attack on my procrastination problem. I'll give you the details then about how I'm going to go about it.

All my experiments must have a clear aim. "Dealing with my procrastination problem" is too vague a goal here. I have to define what specific change I want to make.

What I want to get out of it is very simple. I want to be able to do something many (most?) people do every day, and would never consider it a problem:

I want to write down what I'm going to do the next day, and actually do it.

I am really good at the first half of that. Planning is something I do very well. I have planned the next day (or week) thousands of times. I've taped it to my door or bathroom mirror. I've set alarms, made promises, left trails of instructional sticky notes all through my apartment. But I am not sure if I've ever executed one of these plans all the way through. Honestly, in my 30 years I cannot think of one time I ever did. I will do anything but the 5 to 10 items I thought would be smart ones to do.

It's hard to pinpoint exactly why I'll do anything but what I planned, but it's not that they're necessarily difficult tasks. Sometimes they're so easy that I don't feel any urge to do them right away, and therefore can justifiably do something even easier, like check my email, watch online documentaries, or try a new recipe.

My adversary is the unconscious reactive part of my mind, and by now it's a world-class expert at manipulating me. It's like being a prison guard for Hannibal Lecter. Sure he's locked up, but he's Hannibal Lecter.

So that's my simple, humble dream in life: to list a few things I'd like to get done and go ahead and do them. I could take over the world, if I could only learn to do that. ■

David is the author of Raptitude, a street-level look at the human experience — what makes human beings do what they do, and what that means in real life. He write sabout how to make sense of the earth's most ridiculous animal, how to get better at being one of them.

Reprinted with permission of the original author.
First appeared in *hn.my/procras* (raptitude.com)

The Attack

By SHAANAN COHNEY

I'm a college student studying abroad at the University of Pennsylvania, studying a mixture of CS, Physics, and Music. Eager to learn about the field, I decided to take a course CIS551: Network and Computer Security this semester. This is the story of how as part of the course, I compromised the security of one of my fellow students through social engineering techniques.

For our final project, the class was divided up into two sets of teams, attack and defense. About halfway through the project, defense and attack switched. The role of the defense teams was to construct a secure network chat client. In plain English, they had to write a piece of software that would allow two people to communicate over the internet without fear of wiretapping. The aim of the attack side was to disrupt or compromise their system.

For me the excitement came from the attack side. We had learned in class about "social engineering attacks" as a powerful offensive security tool. The basic premise according to Wiki is "the art of manipulating people into performing actions or divulging confidential information." A trick of a con-man. This was a perfect opportunity for me to actually try putting such a technique into practice whilst still remaining well within the bounds of morality and legality. I asked for permission and was soon granted it. The eagle was a' go.

Phase 1: Information Gathering

First we cross referenced the list of emails on the defense team against the Penn Directory Database. Once we gained full names and school, we cross referenced this against publicly available data using a combination of data mining tools and lookups on social networks such as Facebook and LinkedIn. These were used to build profiles, including photos of potential targets. In our attack proposal we also listed social engineering to warn them of it.

Phase 2: Gaining Rapport/Trust

The next phase of the social engineering attack involved multiple steps. The plan was to place a mole outside the classroom in the engineering building posing as a recruiter from a prestigious company, offering summer internships! First up was obtaining a domain name and email address for use in the attack. We picked X (name redacted) to be the company we would replicate as they are known for being secretive and security focused. We thus registered Xrecruting.com and had the address forward to X.com for authentic looks, while using emails registered to that domain for our purposes.

Next I waited around the engineering buildings looking for a junior administrative assistant or janitor and upon finding one, convinced them that I needed a Penn Lanyard urgently for my senior design presentation as I had forgotten mine. I was soon granted a lanyard and next the team photoshopped an X badge with the face of our "recruiter" (another Penn student) in order to simulate authenticity. We also printed advertising posters to place outside the classroom for further realism. We then placed our mole outside the 551 classroom dressed up in an X t-shirt (purchased online) with the fake badge, the posters, and a laptop set up with a survey. Our representative advertised summer internships in security. A number of students from the class fell for it and entered their information in the survey.

Next we gained further rapport by reaching out to the targets via email. First we initiated contact asking for basic details, a resume, etc.:

> *"This is Joseph from X, we met earlier today. The team and I are very eager to find a candidate that fits our openings here… "*

It wasn't long before our target replied, eager to seize the opportunity:

> *"…please find attached herewith my resume for your kind perusal…I have fair bit of knowledge in Networks and Network Security."*

The game was on, he was falling for it! However, it was one thing to have his trust, but for us to actually use it in some way, we needed to push this further.

Phase 3: Exploitation

To exploit our position of power we had many options, some of which would be pushing the assignment over the edge. With this level of trust it would be feasible to gain access to information protecting online accounts, a very scary thought. However, we decided to go down a different route and instead convinced them of the need to review their source code for recruitment purposes. This allowed us to analyze their code for potential exploits.

> *"My team operates mainly on a Java codebase. Do you have any experience in the area?*
>
> *We'll also get you to submit a few simple coding exercises and perhaps the code from a previous project to see if you're a good fit."*

We exchanged a few more emails back and forth, but it wasn't really getting anywhere. I decided to press a little harder being relatively sure of his trust:

> *"…In looking into specifically which project you would be working on, it would also be good to know if you had any experience in crypto protocols and defensive infrastructure. In regards to this I have two questions. Firstly, is there a professor I could contact in regards to the syllabus and, secondly is there anything that matches this description that you have engaged in as far as you know…?*
> *Could you possibly let me know feasible times in the next week for an interview?*
>
> *Also, are there any current projects in Java you are working on for which a codebase is available for our engineers to review? Even a work in progress is fine. We're really interested in seeing material and your personal projects from this course given the nature of the internship…."*

Finally we struck gold! A few hours later the following appeared in my inbox:

> *"Please find attached herewith 2 Java source code files. (server.java and client.java)*
>
> *These are for a basic chat system application. Further, I and my group would be adding some encryption techniques in it (I'll send you those once we start working on it and progress to some level)."*

And later:

"Hi Joseph,

Please find attached herewith 2 Java code files for a chat system with AES encryption.

Thanks.
Regards."

In the final copy they submitted they had hard coded their AES key, this would be easier than I thought! However this wasn't quite good enough. It would still be difficult to intercept their communication, much less read their messages.

Next I simulated a discussion between the professor and X granting access to the "recruiter" to come visit the demo.

"I have some exciting news and a question for you. I have been informed by Professor Smith that the class has upcoming demos on attack/defense and focusing on network vulnerabilities. I have his permission and now I need yours, to come and watch you demo live…."

"——— Original Message ———Subject: Re: CIS551 Security Recruitment
From: X <X@cis.upenn.edu>
Date: Sun, April 21, 2013 11:41 am

To: "joseph@Xrecruiting.com" <joseph@Xrecruiting.com>

Hi Joseph

I'd be happy to let you and your team come visit my students on Monday during Network Security demos they are undertaking using chat systems they have coded."

The target replied with the affirmative, very eagerly inviting our recruiter in.

"Yes absolutely. You are most welcome. It's this Monday at 4pm in Engineering Building.

Hope to see you there."

"My contact no. is REDACTED if you need any help with location or anything."

Today being demo day, the stage was all set, and our fake recruiter was again in place. I had given her my new Wi-Fi enabled camera to stream a screencap of the enemies messages direct from their screen as they typed, to where my team was sitting a few meters away.

Throughout the demo my team acted as all the other attack teams had, using DDoS, ARP Poisoning and other standard network attacks, to try to compromise their server. However we really had a trump card. Both their encryption key and better yet, the plaintext of their messages.

After launching our usual slew of attacks on their code (most of which worked anyway), we closed the demo and went to meet the other team. When asked if we had any more attacks, I motioned to the recruiter to pass me the camera and as she handed it over, our opponents faces took on stunned looks. It took a good few minutes to convince them of the depth of our attack. Successfully executing this was such an amazing feeling.

I've not yet received my grade for the course, but I feel that more than anything this was a fantastic learning experience before I head out soon to look for a position in industry or for higher study. ■

Shaanan Cohney is a third year Computer Science and Mathematical Physics student studying at the University of Pennsylvania on exchange from the University of Melbourne. In his spare time he is working on a Diploma of Music, and spending many hours toying around with security systems, both physical and digital.

Reprinted with permission of the original author.
First appeared in *hn.my/attack* (shaanan.cohney.info)

mongolab

MongoDB-as-a-Service

AMAZON • AZURE • GOOGLE • JOYENT • RACKSPACE

Power your app with the fastest-growing
cloud database service in the world.